Building Query Compilers
(Under Construction)
[expected time to completion: 5 years]

Guido Moerkotte

September 19, 2006

Contents

I Basics **1**

1 Introduction **3**
1.1 General Remarks . . . 3
1.2 DBMS Architecture . . . 3
1.3 Interpretation versus Compilation . . . 4
1.4 Requirements for a Query Compiler . . . 8
1.5 Search Space . . . 9
1.6 Generation versus Transformation . . . 9
1.7 Focus . . . 10
1.8 Organization of the Book . . . 10

2 Textbook Query Optimization **13**
2.1 Example Query and Outline . . . 13
2.2 Algebra . . . 14
2.3 Canonical Translation . . . 15
2.4 Logical Query Optimization . . . 17
2.5 Physical Query Optimization . . . 24
2.6 Discussion . . . 26

3 Join Ordering **29**
3.1 Queries Considered . . . 29
3.1.1 Query Graph . . . 30
3.1.2 Join Tree . . . 31
3.1.3 Simple Cost Functions . . . 32
3.1.4 Classification of Join Ordering Problems . . . 38
3.1.5 Search Space Sizes . . . 38
3.1.6 Problem Complexity . . . 42
3.2 Deterministic Algorithms . . . 44
3.2.1 Heuristics . . . 44
3.2.2 Determining the Optimal Join Order in Polynomial Time . . . 46
3.2.3 The Maximum-Value-Precedence Algorithm . . . 53
3.2.4 Dynamic Programming . . . 58
3.2.5 Memoization . . . 75
3.2.6 Join Ordering by Generating Permutations . . . 76
3.2.7 A Dynamic Programming based Heuristics for Chain Queries 77
3.2.8 Transformation-Based Approaches . . . 90

3.3 Probabilistic Algorithms . . . 97
3.3.1 Generating Random Left-Deep Join Trees with Cross Products 97
3.3.2 Generating Random Join Trees with Cross Products . . . 98
3.3.3 Generating Random Join Trees without Cross Products . . . 102
3.3.4 Quick Pick . . . 111
3.3.5 Iterative Improvement . . . 112
3.3.6 Simulated Annealing . . . 112
3.3.7 Tabu Search . . . 114
3.3.8 Genetic Algorithms . . . 114
3.4 Hybrid Algorithms . . . 117
3.4.1 Two Phase Optimization . . . 117
3.4.2 AB-Algorithm . . . 117
3.4.3 Toured Simulated Annealing . . . 117
3.4.4 GOO-II . . . 118
3.4.5 Iterative Dynamic Programming . . . 118
3.5 Ordering Order-Preserving Joins . . . 118
3.6 Characterizing Search Spaces . . . 126
3.6.1 Complexity Thresholds . . . 126
3.7 Discussion . . . 129
3.8 Bibliography . . . 129

4 Database Items, Building Blocks, and Access Paths 135
4.1 Disk Drive . . . 135
4.2 Database Buffer . . . 142
4.3 Physical Database Organization . . . 143
4.4 Slotted Page and Tuple Identifier (TID) . . . 146
4.5 Physical Record Layouts . . . 146
4.6 Physical Algebra (Iterator Concept) . . . 148
4.7 Simple Scan . . . 148
4.8 Scan and Attribute Access . . . 149
4.9 Temporal Relations . . . 150
4.10 Table Functions . . . 151
4.11 Indexes . . . 152
4.12 Single Index Access Path . . . 153
4.12.1 Simple Key, No Data Attributes . . . 153
4.12.2 Complex Keys and Data Attributes . . . 159
4.13 Multi Index Access Path . . . 160
4.14 Indexes and Joins . . . 162
4.15 Remarks on Access Path Generation . . . 167
4.16 Counting the Number of Accesses . . . 167
4.16.1 Counting the Number of Direct Accesses . . . 167
4.16.2 Counting the Number of Sequential Accesses . . . 176
4.16.3 Pointers into the Literature . . . 180
4.17 Disk Drive Costs for N Uniform Accesses . . . 181
4.17.1 Number of Qualifying Cylinders, Tracks, and Sectors . . . 181
4.17.2 Command Costs . . . 182
4.17.3 Seek Costs . . . 182

4.17.4 Settle Costs . . . 184
4.17.5 Rotational Delay Costs . . . 184
4.17.6 Head Switch Costs . . . 185
4.17.7 Discussion . . . 185
4.18 Concluding Remarks . . . 187
4.19 Bibliography . . . 187

II Foundations 189

5 Logic, Null, and Boolean Expressions 191
5.1 Two-valued logic . . . 191
5.2 NULL values and two valued logic . . . 191
5.3 Three valued logic . . . 191
5.4 Simplifying Boolean Expressions . . . 191
5.5 Optimizing Boolean Expressions . . . 191
5.6 Bibliography . . . 191

6 An Algebra for Sets, Bags, and Sequences 195
6.1 Sets, Bags, and Sequences . . . 195
6.1.1 Sets . . . 195
6.1.2 Duplicate Data: Bags . . . 196
6.1.3 Ordered Data: Sequences . . . 198
6.2 Algebra . . . 200
6.2.1 The Operators . . . 200
6.2.2 Preliminaries . . . 201
6.2.3 Operator Signatures . . . 203
6.2.4 Selection . . . 205
6.2.5 Projection . . . 205
6.2.6 Map . . . 205
6.2.7 Join Operators . . . 205
6.2.8 Linearity and Reorderability . . . 207
6.2.9 Reordering of joins and outer-joins . . . 210
6.2.10 Basic Equivalences for d-Join and Grouping . . . 213
6.2.11 Simplifying Expressions Containing Joins . . . 214
6.2.12 Reordering Joins and Grouping . . . 215
6.2.13 ToDo . . . 223
6.3 Logical Algebra for Sequences . . . 223
6.3.1 Introduction . . . 223
6.3.2 Algebraic Operators . . . 224
6.3.3 Equivalences . . . 227
6.4 Bibliography . . . 227
6.5 Literature . . . 227

7 Declarative Query Representation 229
7.1 Calculus Representations . . . 229
7.2 Datalog . . . 229
7.3 Tableaux Representation . . . 229
7.4 Monoid Comprehension . . . 229
7.5 Expressiveness . . . 229
7.6 Bibliography . . . 229

8 Query Equivalence, Containment, Minimization, and Factorization 231
8.1 Set Semantics . . . 231
8.1.1 Conjunctive Queries . . . 231
8.1.2 . . . with Inequalities . . . 234
8.1.3 . . . with Negation . . . 235
8.1.4 . . . under Constraints . . . 235
8.1.5 . . . with Aggregation . . . 235
8.2 Bag Semantics . . . 235
8.2.1 Conjunctive Queries . . . 235
8.3 Sequences . . . 235
8.3.1 Path Expressions . . . 235
8.4 Minimization . . . 236
8.5 Detecting common subexpressions . . . 237
8.5.1 Simple Expressions . . . 237
8.5.2 Algebraic Expressions . . . 237
8.6 Bibliography . . . 237

9 Translation and Lifting 239
9.1 Query Language to Calculus . . . 239
9.2 Query Language to Algebra . . . 239
9.3 Calculus to Algebra . . . 239
9.4 Algebra to Calculus . . . 239
9.5 Bibliography . . . 239

10 Functional Dependencies 241
10.1 Functional Dependencies . . . 241
10.2 Functional Dependencies in the presence of NULL values . . . 241
10.3 Deriving Functional Dependencies over algebraic operators . . . 241
10.4 Bibliography . . . 241

III Enabling Techniques 243

11 Simple Rewrites 245
11.1 Simple Adjustments . . . 245
11.1.1 Rewriting Simple Expressions . . . 245
11.1.2 Normal forms for queries with disjunction . . . 247
11.2 Deriving new predicates . . . 247
11.2.1 Collecting conjunctive predicates . . . 247

11.2.2 Equality . . . 247
11.2.3 Inequality . . . 248
11.2.4 Aggregation . . . 249
11.2.5 ToDo . . . 251
11.3 Eliminating Redundant Joins . . . 251
11.4 Distinct Pull-Up and Push-Down . . . 251
11.5 Set-Valued Attributes . . . 251
11.5.1 Introduction . . . 251
11.5.2 Preliminaries . . . 252
11.5.3 Query Rewrite . . . 252
11.6 Bibliography . . . 253

12 View Merging **257**
12.1 View Resolution . . . 257
12.2 Simple View Merging . . . 257
12.3 Predicate Move Around (Predicate pull-up and push-down) . . . 258
12.4 Complex View Merging . . . 259
12.4.1 Views with Distinct . . . 259
12.4.2 Views with Group-By and Aggregation . . . 260
12.4.3 Views in IN predicates . . . 261
12.4.4 Final Remarks . . . 261
12.5 Bibliography . . . 262

13 Unnesting Nested Queries **263**
13.1 Classification of nested queries . . . 263
13.2 Queries of Type A . . . 264
13.3 Queries of Type N . . . 265
13.4 Queries of Type J . . . 268
13.5 Queries of Type JA . . . 270
13.6 Alternative locations . . . 271
13.7 Different Kinds of Dependency . . . 273
13.8 Unnesting IN . . . 275
13.9 Further reading . . . 275
13.10 History . . . 275
13.11 Bibliography . . . 276
13.12 ToDo . . . 276

14 Optimizing Queries with Materialized Views **277**
14.1 Conjunctive Views . . . 277
14.2 Views with Grouping and Aggregation . . . 277
14.3 Views with Disjunction . . . 277
14.4 Bibliography . . . 277

15 Semantic Query Rewrite **279**
15.1 Constraints and their impact on query optimization . . . 279
15.2 Semantic Query Rewrite . . . 279
15.3 Exploiting Uniqueness in Query Optimization . . . 280

15.4 Bibliography 280

IV Search Space Limits and Extensions 281

16 Current Search Space and Its Limits 283
16.1 Plans with Outer Joins, Semijoins and Antijoins 283
16.2 Expensive Predicates and Functions 283
16.3 Techniques to Reduce the Search Space 283
16.4 Bibliography 283

17 Quantifier treatment 285
17.1 Pseudo-Quantifiers 285
17.2 Existential quantifier 286
17.3 Universal quantifier 286
17.4 Bibliography 289

18 Optimizing Queries with Disjunctions 291
18.1 Introduction 291
18.2 Using Disjunctive or Conjunctive Normal Forms 292
18.3 Bypass Plans 292
18.4 Implementation remarks 293
18.5 Other plan generators/query optimizer 294
18.6 Bibliography 294

19 Grouping and Aggregation 295
19.1 Introduction 295
19.2 Aggregate Functions 296
19.3 Normalization and Translation 299
19.3.1 Grouping and Aggregation in Different Query Languages . . 299
19.3.2 Extensions to Algebra 300
19.3.3 Normalization 300
19.3.4 Translation 300
19.4 Lazy and eager group by 300
19.5 Coalescing Grouping 302
19.6 More Possibilities 305
19.6.1 **Eager/Lazy Group-By-Count** 307
19.7 Translation into Our Notation 312
19.8 Aggregation of Non-Equi-Join Results 313
19.9 Bibliography 313

20 Grouping and Aggregation 315
20.1 Introduction 315
20.2 Lazy and eager group by 316
20.3 Coalescing Grouping 319
20.4 ToDo 322

V Plan Generation 325

21 Introduction to Plan Generation 327
21.1 Search Space Selection . . . 327
21.2 Complexity Results . . . 327
21.3 Implementation Approaches and Architectures . . . 327
21.3.1 Hard-wired Algorithms . . . 327
21.3.2 Rule Based Approaches . . . 327
21.3.3 Blackboard Architecture . . . 328
21.4 Index Selection . . . 328
21.5 Disjunctive Queries . . . 328
21.6 Outer Joins . . . 328
21.7 Plan Improvements, Postprocessing, and Polishing . . . 328
21.7.1 Pushing group operators . . . 328
21.7.2 Predicate pull-up . . . 329
21.7.3 Polishing . . . 329
21.7.4 Optimizing complex boolean expressions . . . 329
21.8 Bibliography . . . 330

22 Hard-Wired Algorithms 331
22.1 Hard-wired Dynamic Programming . . . 331
22.1.1 Introduction . . . 331
22.1.2 A plan generator for bushy trees . . . 334
22.1.3 A plan generator for bushy trees and expensive selections . . . 335
22.1.4 A plan generator for bushy trees, expensive selections and functions . . . 337
22.2 Bibliography . . . 337

23 Rule-Based Algorithms 339
23.1 Rule-based Dynamic Programming . . . 339
23.2 Rule-based Memoization . . . 339
23.3 Bibliography . . . 339

24 Deriving and Dealing with Interesting Orderings and Groupings 341
24.1 Introduction . . . 341
24.2 Problem Definition . . . 342
24.2.1 Ordering . . . 342
24.2.2 Grouping . . . 344
24.2.3 Functional Dependencies . . . 345
24.2.4 Algebraic Operators . . . 345
24.2.5 Plan Generation . . . 346
24.3 Overview . . . 347
24.4 Detailed Algorithm . . . 350
24.4.1 Overview . . . 350
24.4.2 Determining the Input . . . 351
24.4.3 Constructing the NFSM . . . 351
24.4.4 Constructing the DFSM . . . 354

24.4.5 Precomputing Values . . . 354
24.4.6 During Plan Generation . . . 355
24.4.7 Reducing the Size of the NFSM . . . 356
24.4.8 Complex Ordering Requirements . . . 359
24.5 Converting a NFSM into a DFSM . . . 360
24.5.1 Definitions . . . 360
24.5.2 The Transformation Algorithm . . . 361
24.5.3 Correctness of the FSM Transformation . . . 361
24.6 Experimental Results . . . 362
24.7 Total Impact . . . 362
24.8 Influence of Groupings . . . 364
24.9 Annotated Bibliography . . . 366

25 Other Issues in Plan Generation 371
25.1 Plan Generation for Compressed Databases . . . 371
25.2 Generating DAGs-Plans . . . 371

VI Cardinality and Cost Estimates 373

26 Introduction 375
26.1 Selection . . . 376
26.2 Join . . . 377
26.3 Projection, Grouping, and Duplicate Elimination . . . 377
26.4 Feedback from Runtime . . . 378
26.5 Bibliography . . . 378

27 Statistics and Cardinality Estimates 381
27.1 Uniformity and Independence Assumption . . . 381
27.2 Dropping the Uniformity Assumption . . . 382
27.2.1 ToDo . . . 382
27.3 Dropping the Independence Assumption . . . 382
27.4 Bibliography . . . 382

28 Cost functions for selected algebraic operators 383
28.1 Scan Operations . . . 383
28.2 I/O costs for index-based access . . . 383
28.3 I/O costs for join algorithms . . . 384
28.4 Sorting, Grouping, and Duplicate Elimination . . . 386
28.5 Bibliography . . . 386

VII Implementation 389

29 Architecture of a Query Compiler 391
29.1 Compilation process . . . 392
29.2 Architecture . . . 392
29.3 Control Blocks . . . 392

29.4 Memory Management . . . 394
29.5 Tracing and Plan Visualization . . . 394
29.6 Driver . . . 394
29.7 Bibliography . . . 394

30 Internal Representations 395
30.1 Requirements . . . 395
30.2 Algebraic Representations . . . 395
30.2.1 Graph Representations . . . 396
30.2.2 Query Graph . . . 396
30.2.3 Operator Graph . . . 396
30.3 Query Graph Model (QGM) . . . 396
30.4 Classification of Predicates . . . 396
30.5 Treatment of Distinct . . . 396
30.6 Query Analysis and Materialization of Analysis Results . . . 396
30.7 Query and Plan Properties . . . 397
30.8 Conversion to the Internal Representation . . . 399
30.8.1 Preprocessing . . . 399
30.8.2 Translation into the Internal Representation . . . 399
30.9 Bibliography . . . 399

31 Details on the Phases of Query Compilation 401
31.1 Parsing . . . 401
31.2 Semantic Analysis, Normalization, Factorization, Constant Folding, and Translation . . . 401
31.3 Normalization . . . 403
31.4 Factorization . . . 403
31.5 Constant Folding . . . 404
31.6 Semantic analysis . . . 404
31.7 Translation . . . 406
31.8 Rewrite I . . . 411
31.9 Plan Generation . . . 411
31.10 Rewrite II . . . 411
31.11 Code generation . . . 411
31.12 Bibliography . . . 412

32 Quality Assurance 413
32.1 Verification . . . 413
32.2 Validation . . . 413
32.3 Debugging . . . 413
32.4 Test Data Generation . . . 413
32.5 Benchmarking . . . 413
32.6 Bibliography . . . 413

VIII Selected Topics **415**

33 Generating Plans for Top-N-Queries? **417**
33.1 Motivation and Introduction 417
33.2 Optimizing for the First Tuple 417
33.3 Optimizing for the First N Tuples 417

34 Recursive Queries **419**

35 Issues Introduced by OQL **421**
35.1 Type-Based Rewriting and Pointer Chasing Elimination 421
35.2 Class Hierarchies 423
35.3 Cardinalities and Cost Functions 425

36 Issues Introduced by XPath **427**
36.1 A Naive XPath-Interpreter and its Problems 427
36.2 Dynamic Programming and Memoization 427
36.3 Naive Translation of XPath to Algebra 427
36.4 Pushing Duplicate Elimination 427
36.5 Avoiding Duplicate Work 427
36.6 Avoiding Duplicate Generation 427
36.7 Index Usage and Materialized Views 427
36.8 Cardinalities and Costs 427
36.9 Bibliography 427

37 Issues Introduced by XQuery **429**
37.1 Reordering in Ordered Context 429
37.2 Result Construction 429
37.3 Unnesting Nested XQueries 429
37.4 Cardinalities and Cost Functions 429
37.5 Bibliography 429

38 Outlook **431**

A Query Languages? **433**
A.1 Designing a query language 433
A.2 SQL 433
A.3 OQL 433
A.4 XPath 433
A.5 XQuery 433
A.6 Datalog 433

B Query Execution Engine (?) **435**

C Glossary of Rewrite and Optimization Techniques **437**

D Example Query Compiler **443**
D.1 Research Prototypes . . . 443
D.1.1 AQUA and COLA . . . 443
D.1.2 Black Dahlia II . . . 443
D.1.3 Epoq . . . 443
D.1.4 Ereq . . . 445
D.1.5 Exodus/Volcano/Cascade . . . 446
D.1.6 Freytags regelbasierte System R-Emulation . . . 448
D.1.7 Genesis . . . 449
D.1.8 GOMbgo . . . 451
D.1.9 Gral . . . 454
D.1.10 Lambda-DB . . . 457
D.1.11 Lanzelotte in short . . . 457
D.1.12 Opt++ . . . 458
D.1.13 Postgres . . . 458
D.1.14 Sciore & Sieg . . . 460
D.1.15 Secondo . . . 460
D.1.16 Squiral . . . 460
D.1.17 System R and System R* . . . 462
D.1.18 Starburst and DB2 . . . 462
D.1.19 Der Optimierer von Straube . . . 465
D.1.20 Other Query Optimizer . . . 466
D.2 Commercial Query Compiler . . . 468
D.2.1 The DB 2 Query Compiler . . . 468
D.2.2 The Oracle Query Compiler . . . 468
D.2.3 The SQL Server Query Compiler . . . 471

E Some Equalities for Binomial Coefficients **473**

Bibliography **475**

Index **537**

F ToDo **539**

List of Figures

1.1 DBMS architecture 4
1.2 Query interpreter 4
1.3 Simple query interpreter 5
1.4 Query compiler 5
1.5 Query compiler architecture 6
1.6 Execution plan 7
1.7 Potential and actual search space 10
1.8 Generation vs. transformation 11

2.1 Relational algebra 15
2.2 Equivalences for the relational algebra 16
2.3 (Simplified) Canonical translation of SQL to algebra 17
2.4 Text book query optimization 18
2.5 Logical query optimization 19
2.6 Different join trees 21
2.7 Plans for example query (Part I) 23
2.8 Plans for example query (Part II) 24
2.9 Physical query optimization 26
2.10 Plan for example query after physical query optimization 27

3.1 Query graph for example query of Section 2.1 30
3.2 Query graph shapes 31
3.3 Illustrations for the IKKBZ Algorithm 52
3.4 A query graph, its directed join graph, some spanning trees and join trees 54
3.5 A query graph, its directed join tree, a spanning tree and its problem . 55
3.6 Search space with sharing under optimality principle 60
3.7 Algorithm `DPsize` 67
3.8 Algorithm `DPsub` 69
3.9 Size of the search space for different graph structures 70
3.10 Algorithm `DPccp` 71
3.11 Enumeration Example for `DPccp` 72
3.12 Sample graph to illustrate `EnumerateCsgRec` 73
3.13 Call sequence for Figure 3.12 74
3.14 Example of rule transformations (RS-1) 95
3.15 Encoding Trees 100
3.16 Paths 101

3.17 Tree-merge . . . 104
3.18 Algorithm `UnrankDecomposition` . . . 105
3.19 Leaf-insertion . . . 106
3.20 A query graph, its tree, and its standard decomposition graph . . . 107
3.21 Algorithm `Adorn` . . . 109
3.22 A query graph, a join tree, and its encoding . . . 116
3.23 Pseudo code for `IDP-1` . . . 119
3.24 Pseudocode for IDP-2 . . . 120
3.25 Subroutine `applicable-predicates` . . . 122
3.26 Subroutine `construct-bushy-tree` . . . 122
3.27 Subroutine `extract-plan` and its subroutine . . . 123
3.28 Impact of selectivity on the search space . . . 130
3.29 Impact of relation sizes on the search space . . . 131
3.30 Impact of parameters on the performance of heuristics . . . 132
3.31 Impact of selectivities on probabilistic procedures . . . 133

4.1 Disk drive assembly . . . 136
4.2 Disk drive read request processing . . . 137
4.3 Time to read 100 MB from disk (depending on the number of 8 KB blocks read at once) . . . 140
4.4 Time needed to read n random pages . . . 141
4.5 Break-even point in fraction of total pages depending on page size . . . 141
4.6 Physical organization of a relational database . . . 144
4.7 Slotted pages and TIDs . . . 146
4.8 Various physical record layouts . . . 147
4.9 Clustered vs. non-clustered index . . . 153
4.10 Illustration of seek cost estimate . . . 182

5.1 Truth tables for two-valued logic . . . 191
5.2 Simplification Rules . . . 192
5.3 Laws for two-valued logic . . . 193
5.4 Truth tables for three-valued logic . . . 194

6.1 Laws for Set Operations . . . 195
6.2 Laws for Bag Operations . . . 197
6.3 Outer join examples . . . 211
6.4 Two equivalent plans . . . 217
6.5 Applications of coalescing grouping . . . 223
6.6 Example for Map Operator . . . 225
6.7 Examples for Unary and Binary $\hat{\Gamma}$ . . . 226

11.1 Simplification rules for boolean expressions . . . 248
11.2 Axioms for equality . . . 248
11.3 . . . 255
11.4 Axioms for inequality . . . 256

18.1 DNF plans . . . 292
18.2 CNF plans . . . 293

18.3 Bypass plans 293

19.1 Two equivalent plans 297
19.2 Applications of coalescing grouping 306

20.1 Two equivalent plans 317
20.2 Applications of coalescing grouping 323

21.1 Early grouping 328

22.1 A sample execution plan 332
22.2 Different join operator trees 333
22.3 Bottom up plan generation 335
22.4 A Dynamic Programming Optimization Algorithm 336

24.1 Propagation of orderings and groupings 346
24.2 Possible FSM for orderings 347
24.3 Possible FSM for groupings 348
24.4 Combined FSM for orderings and groupings 349
24.5 Possible DFSM for Figure 24.4 349
24.6 Preparation steps of the algorithm 350
24.7 Initial NFSM for sample query 352
24.8 NFSM after adding D_{FD} edges 353
24.9 NFSM after pruning artificial states 353
24.10 Final NFSM 354
24.11 Resulting DFSM 354
24.12 *contains* Matrix 354
24.13 *transition* Matrix 355
24.14 Plan generation for different join graphs, Simmen's algorithm (left) vs. our algorithm (middle) 362
24.15 Memory consumption in KB for Figure 24.14 364
24.16 Time requirements for the preparation step 367
24.17 Space requirements for the preparation step 367

29.1 The compilation process 391
29.2 Class Architecture of the Query Compiler 392
29.3 Control Block Structure 393

31.1 Expression 402
31.2 Expression hierarchy 403
31.3 Expression 404
31.4 Query 1 405
31.5 Internal representation 407
31.6 An algebraic operator tree with a d-join 410
31.7 Algebra 410

35.1 Algebraic representation of a query 421
35.2 A join replacing pointer chasing 423
35.3 A Sample Class Hierarchy 424

35.4 Implementation of Extents 425

D.1 Beispiel einer Epoq-Architektur 444
D.2 Exodus Optimierer Generator 446
D.3 Organisation der Optimierung 449
D.4 Ablauf der Optimierung 452
D.5 Architektur von GOMrbo 453
D.6 a) Architektur des Gral-Optimierers; b) Operatorhierarchie nach Kosten 454
D.7 Die Squiralarchitektur 461
D.8 Starburst Optimierer 463
D.9 Der Optimierer von Straube 465

Preface

Goals

Primary Goals:

- book covers many query languages (at least SQL, OQL, XQuery (XPath))
- techniques should be represented as query language independent as possible
- book covers all stages of the query compilation process
- book completely covers fundamental issues
- book gives implementation details and tricks

Secondary Goals:

- book is thin
- book is not totally unreadable
- book separates concepts from implementation techniques

Organizing the material is not easy: The same topic pops up

- at different stages of the query compilation process and
- at different query languages

Acknowledgements

Introducer to query optimization: Günther von Bültzingsloewen
Peter Lockemann
First paper coauthor: Stefan Karl,
Coworkers: Alfons Kemper, Klaus Peithner, Michael Steinbrunn, Donald Kossmann, Carsten Gerlhof, Jens Claussen,
Sophie Cluet, Vassilis Christophides, Georg Gottlob, V.S. Subramanian,
Sven Helmer, Birgitta König-Ries, Wolfgang Scheufele, Carl-Christian Kanne, Thomas Neumann, Norman May, Matthias Brantner
Robin Aly
Discussions: Umesh Dayal, Dave Maier, Gail Mitchell, Stan Zdonik, Tamer Özsu, Arne Rosenthal,

Don Chamberlin, Bruce Lindsay, Guy Lohman, Mike Carey, Bennet Vance, Laura Haas, Mohan, CM Park,

Yannis Ioannidis, Götz Graefe, Serge Abiteboul, Claude Delobel Patrick Valduriez, Dana Florescu, Jerome Simeon, Mary Fernandez, Christoph Koch, Adam Bosworth, Joe Hellerstein, Paul Larson, Hennie Steenhagen, Harald Schöning, Bernhard Seeger,

Encouragement: Anand Deshpande

Manuscript: Simone Seeger,

and many others to be inserted.

Part I

Basics

Chapter 1

Introduction

1.1 General Remarks

Query languages like SQL or OQL are declarative. That is, they specify what the user wants to retrieve and not how to retrieve it. It is the task of the query compiler to generate a *query evaluation plan* (*evaluation plan* for short, or *execution plan* or simply *plan*) for a given query. The query evaluation plan (QEP) essentially is an operator tree with physical algebraic operators as nodes. It is evaluated by the runtime system. Figure 1.6 shows a detailed execution plan ready to be interpreted by the runtime system. Figure 22.1 shows an abstraction of a query plan often used to explain algorithms or optimization techniques.

The book tries to demystify query optimization and query optimizers. By means of the multi-lingual query optimizer BD II, the most important aspects of query optimizers and their implementation are discussed. We concentrate not only on the *query optimizer* core (Rewrite I, Plan Generator, Rewrite II) of the query compilation process but touch on all issues from parsing to code generation and quality assurance.

We start by giving a two-module overview of a database management system. One of these modules covers the functionality of the query compiler. The query compiler itself involves several submodules. For each submodule we discuss the features relevant for query compilation.

1.2 DBMS Architecture

Any database management system (DBMS) can be divided into two major parts: the compile time system (CTS) and the runtime system (RTS). User commands enter the compile time system which translates them into executables which are then interpreted by the runtime system (Fig. 1.1).

The input to the CTS are statements of several kinds, for example connect to a database (starts a session), disconnect from a database, create a database, drop a database, add/drop a schema, perform schema changes (add relations, object types, constraints, ...), add/drop indexes, run statistics commands, manually modify the DBMS statistics, begin of a transaction, end of a transaction, add/drop a view, update database items (e.g. tuples, relations, objects), change authorizations, and state a query. Within the book, we will only be concerned with the tiny last item.

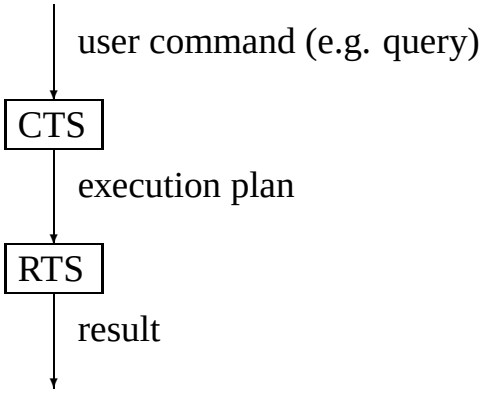


Figure 1.1: DBMS architecture

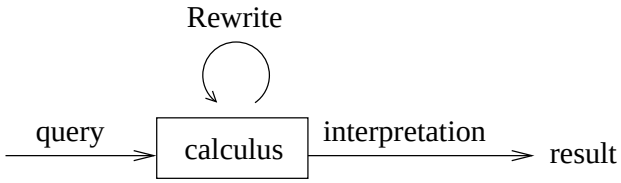


Figure 1.2: Query interpreter

1.3 Interpretation versus Compilation

There are two essential approaches to process a query: *interpretation* and *compilation*.

The path of a query through a query interpreter is illustrated in Figure 1.2. Query interpretation translates the query string into some internal representation that is mostly calculus-based. Optionally, some rewrite on this representation takes place. Typical steps during this rewrite phase are unnesting nested queries, pushing selections down, and introducing index structures. After that, the query is interpreted. A simple query interpreter is sketched in Figure 1.3. The first function, `interprete`, takes a simple SQL block and extracts the different clauses, initializes the result `R` and calls `eval`. Then, `eval` recursively evaluates the query by first producing the cross product of the entries in the `from` clause. After all of them have been processed, the predicate is applied and for those tuples where the `where` predicate evaluates to true, a result tuple is constructed and added to the result set `R`. Obviously, the sketeched interpreter is far from being efficient. A much better approach has been described by Wong and Youssefi [895, 927].

Let us now discuss the compilation approach. The different steps are summarized in Figure 1.4. First, the query is rewritten. Again, unnesting nested queries is a main technique for performance gains. Other rewrites will be discussed in Part III. After the rewrite, the plan generation takes place. Here, an optimal plan is constructed. Whereas typically rewrite takes place on a calculus-based representation of the query, plan generation constructs an algebraic expression containing well-known operators like selection and join. Sometimes, after plan generation, the generated plan is refined: some polishing takes place. Then, code is generated, that can be interpreted by the runtime system. More specifically, the query execution engine—a part of the runtime

```
interprete(SQLBlock x) {

        /* possible rewrites go here */
        s := x.select();
        f := x.from();
        w := x.where();
        R := ∅; /* result */
        t := []; /* empty tuple */
        eval(s, f, w, t, R);
        return R;
}


eval(s, f, w, t, R) {

        if(f.empty())
            if(w(t))
                R += s(t);
        else
            foreach(t' ∈ first(f))
                eval(s, tail(f), w, t ∘ t', R);
}
```

Figure 1.3: Simple query interpreter

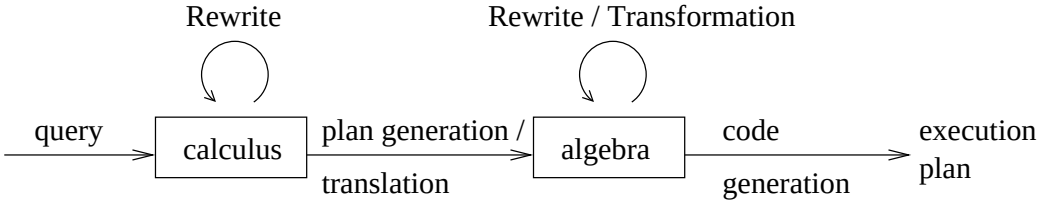


Figure 1.4: Query compiler

system—interpretes the query execution plan. Let us illustrate this. The following query is Query 1 of the now obsolete TPC-D benchmark [843].

```
SELECT    RETURNFLAG, LINESTATUS,
          SUM(QUANTITY) as SUM_QTY,
          SUM(EXTENDEDPRICE) as SUM_EXTPR,
          SUM(EXTENDEDPRICE * (1 - DISCOUNT)),
          SUM(EXTENDEDPRICE * (1 - DISCOUNT)*
             (1 + TAX)),
          AVG(QUANTITY),
          AVG(EXTENDEDPRICE),
          AVG(DISCOUNT),
```

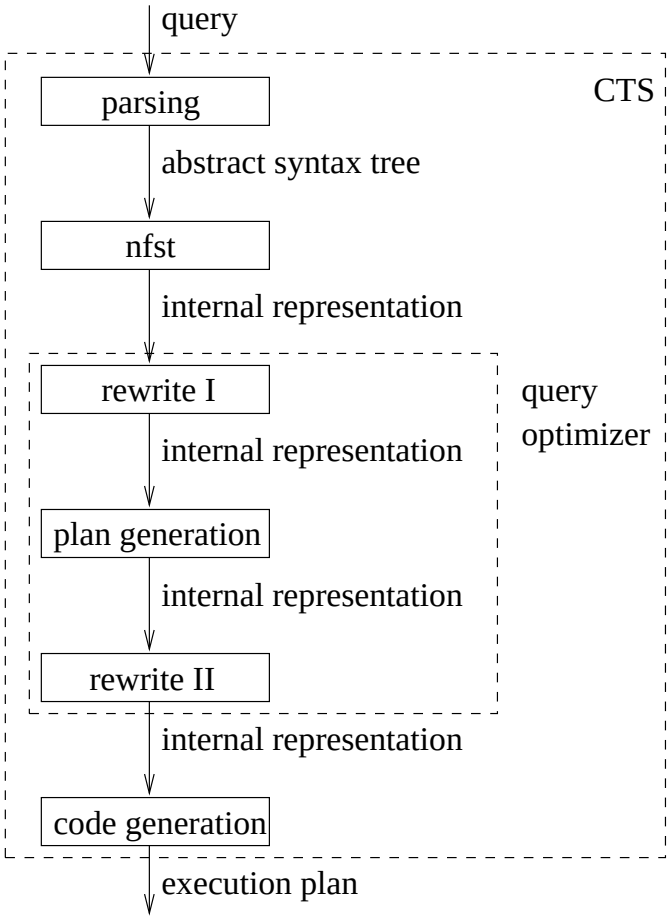


Figure 1.5: Query compiler architecture

```
          COUNT(*)
FROM      LINEITEM
WHERE     SHIPDDATE <= DATE '1998-12-01'
GROUP BY  RETURNFLAG, LINESTATUS
ORDER BY  RETURNFLAG, LINESTATUS
```

The CTS translates this query into a query execution plan. Part of the plan is shown in Fig. 1.6. One rarely sees a query execution plan. This is the reason why I included one. But note that the form of query execution plans differs from DBMS to DBMS since it is (unfortunately) not standardized the way SQL is. Most DBMSs can give the user abstract representations of query plans. It is worth the time to look at the plans generated by some commercial DBMSs.

I do not expect the reader to understand the plan in all details. Some of these details will become clear later. Anyway, this plan is given to the RTS which then interprets it. Part of the result of the interpretation might look like this:

```
(group
  (tbscan
     {segment 'lineitem.C4Kseg' 0 4096}
     {nalslottedpage 4096}
     {ctuple 'lineitem.cschema'}
     [ 20
       LOAD_PTR                    1
       LOAD_SC1_C    8           1  2 // L_RETURNFLAG
       LOAD_SC1_C    9           1  3 // L_LINESTATUS
       LOAD_DAT_C   10           1  4 // L_SHIPDATE
       LEQ_DAT_ZC_C 4 '1998-02-09'  1
     ] 2 1  // number of help-registers and selection-register
  ) 10 22 // hash table size,  number of registers
  [ // init
    MV_UI4_C_C    1              0    // COUNT(*) = 0
    LOAD_SF8_C    4           1  6    // L_QUANTITY
    LOAD_SF8_C    5           1  7    // L_EXTENDEDPRICE
    LOAD_SF8_C    6           1  8    // L_DISCOUNT
    LOAD_SF8_C    7           1  9    // L_TAX
    MV_SF8_Z_C    6             10    // SUM/AVG(L_QUANTITY)
    MV_SF8_Z_C    7             11    // SUM/AVG(L_EXTENDEDPRICE)
    MV_SF8_Z_C    8             12    // AVG(L_DISCOUNT)
    SUB_SF8_CZ_C  1.0          8 13    // 1 - L_DISCOUNT
    ADD_SF8_CZ_C  1.0          9 14    // 1 + L_TAX
    MUL_SF8_ZZ_C  7           13 15    // SUM(L_EXTDPRICE * (1 - L_DISC))
    MUL_SF8_ZZ_C 15           14 16    // SUM((...) * (1 + L_TAX))
  ] [ // advance
    INC_UI4       0                    // inc COUNT(*)
    MV_PTR_Y      1              1
    LOAD_SF8_C    4           1  6    // L_QUANTITY
    LOAD_SF8_C    5           1  7    // L_EXTENDEDPRICE
    LOAD_SF8_C    6           1  8    // L_DISCOUNT
    LOAD_SF8_C    7           1  9    // L_TAX
    MV_SF8_Z_A    6             10    // SUM/AVG(L_QUANTITY)
    MV_SF8_Z_A    7             11    // SUM/AVG(L_EXTENDEDPRICE)
    MV_SF8_Z_A    8             12    // AVG(L_DISCOUNT)
    SUB_SF8_CZ_C  1.0          8 13    // 1 - L_DISCOUNT
    ADD_SF8_CZ_C  1.0          9 14    // 1 + L_TAX
    MUL_SF8_ZZ_B  7           13 17 15 // SUM(L_EXTDPRICE * (1 - L_DISC))
    MUL_SF8_ZZ_A 17           14 16    // SUM((...) * (1 + L_TAX))
  ] [ // finalize
    UIFC_C         0             18
    DIV_SF8_ZZ_C  10          18 19    // AVG(L_QUANTITY)
    DIV_SF8_ZZ_C  11          18 20    // AVG(L_EXTENDEDPRICE)
    DIV_SF8_ZZ_C  12          18 21    // AVG(L_DISCOUNT)
  ] [ // hash program
    HASH_SC1  2 HASH_SC1 3
  ] [ // compare program
    CMPA_SC1_ZY_C 2           2  0
    EXIT_NEQ      0
    CMPA_SC1_ZY_C 3           3  0
  ])
```

Figure 1.6: Execution plan

RETURNFLAG	LINESTATUS	SUM_QTY	SUM_EXTPR	...
A	F	3773034	5319329289.68	...
N	F	100245	141459686.10	...
N	O	7464940	10518546073.98	...
R	F	3779140	5328886172.99	...

This should look familar to you.

The above query plan is very simple. It contains only a few algebraic operators. Usually, more algebraic operators are present and the plan is given in a more abstract form that cannot be directly executed by the runtime system. Fig. 2.10 gives an example of an abstracted more complex operator tree. We will work with representations closer to this one.

A typical query compiler architecture is shown in Figure 1.5. The first component is the parser. It produces an abstract syntax tree. This is not always the case but this intermediate representation very much simplifies the task of following component. The NFST component performs several tasks. The first step is normalization. This mainly deals with introducing new variables for subexpressions. Factorization and semantic analysis are performed during NFST. Last, the abstract syntax tree is translated into the internal representation. All these steps can typically be performed during a single path through the query representation. Semantic analysis requires looking up schema definitions. This can be expensive and, hence, the number of lookups should be minimized. After NFST the core optimization steps rewrite I and plan generation take place. Rewrite II does some polishing before code generation. These modules directly correspond to the phases in Figure 1.4. They are typically further devided into submodules handling subphases. The most prominent example is the preparation phase that takes place just before the actual plan generation takes place. In our figures, we think of preparation as being part of the plan generation.

1.4 Requirements for a Query Compiler

Here are the main requirements for a query compiler:

1. Correctness
2. Completeness
3. Generate optimal plan (viz avoid the worst case)
4. Efficiency, generate the plan fast, do not waste memory
5. Graceful degradation
6. Robustness

First of all, the query compiler must produce correct query evaluation plans. That is, the result of the query evaluation plan must be the result of the query as given by the specification of the query language. It must also cover the complete query language. The next issue is that an optimal query plan must (should) be generated. However, this is not always that easy. That is why some database researchers say that one must avoid the worst plan. Talking about the quality of a plan requires us to fix the optimization goal. Several goals are reasonable: We can optimize throughput, minimize response time, minimize resource consumption (both, memory and CPU), and so on. A good query compiler supports two optimization goals: minimize resource consumption and minimize the time to produce the first tuple. Obviously, both goals cannot be achieved

at the same time. Hence, the query compiler must be instructed about the optimization goal.

Irrespective of the optimization goal, the query compiler should produce the query evaluation plan fast. It does not make sense to take 10 seconds to optimize a query whose execution time is below a second. This sounds reasonable but is not trivial to achieve. As we will see, the number of query execution plans that are equivalent to a given query, i.e. produce the same result as the query, can be very large. Sometimes, very large even means that not all plans can be considered. Taking the wrong approach to plan generation will result in no plan at all. This is the contrary of graceful degradation. Expressed positively, graceful degradation means that in case of limited resources, a plan is generated that may not be the optimal plan, but also not that far away from the optimal plan.

Last, typical software quality criteria should be met. We only mentioned robustness in our list, but others like maintainability must be met also.

1.5 Search Space

For a given query, there typically exists a high number of plans that are equivalent to the plan. Not all of these plans are accessible. Only those plans that can be generated by known optimization techniques (mainly algebraic equivalences) can potentially be generated. Since this number may still be too large, many query compilers restrict their search space further. We call the search space explored by a query optimizer the *actual search space*. The *potential search space* is the set of all plans that are known to be equivalent to the given query by applying the state of the art of query optimization techniques. The whole set of plans equivalent to a given query is typically unknown: we are not sure whether all optimization techniques have been discovered so far. Figure 1.7 illustrates the situation. Note that we run into problems if the actual search space is not a subset of the equivalent plans. Then the query compiler produces wrong results. As we will see in several chapters of this book, some optimization techniques have been proposed that produce plans that are not equivalent to the original query.

1.6 Generation versus Transformation

Two different approaches to plan generation can be distinguished:

- The transformation-based approach transforms one query execution plan into another equivalent one. This can, for example, happen by applying an algebraic equivalence to a query execution plan in order to yield a better plan.
- The generic or synthetic approach produces a query execution plan by assembling building blocks and adding one algebraic operator after the other, until a complete query execution plan has been produced. Note that in this approach only when all building blocks and algebraic opertors have been introduced the internal representation can be executed. Before that, no (complete) plan exists.

For an illustration see Figure 1.8.

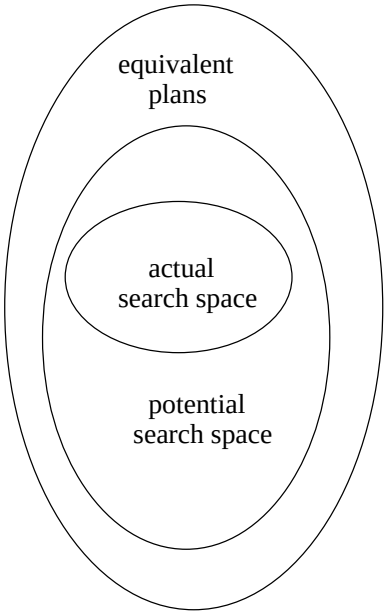


Figure 1.7: Potential and actual search space

A very important issue is how to explore the search space. Several well-known approaches exist: A*, Branch-and-bound, greedy algorithms, hill-climbing, dynamic programming, memoization, [206, 513, 514, 654]. These form the basis for most of the plan generation algorithms.

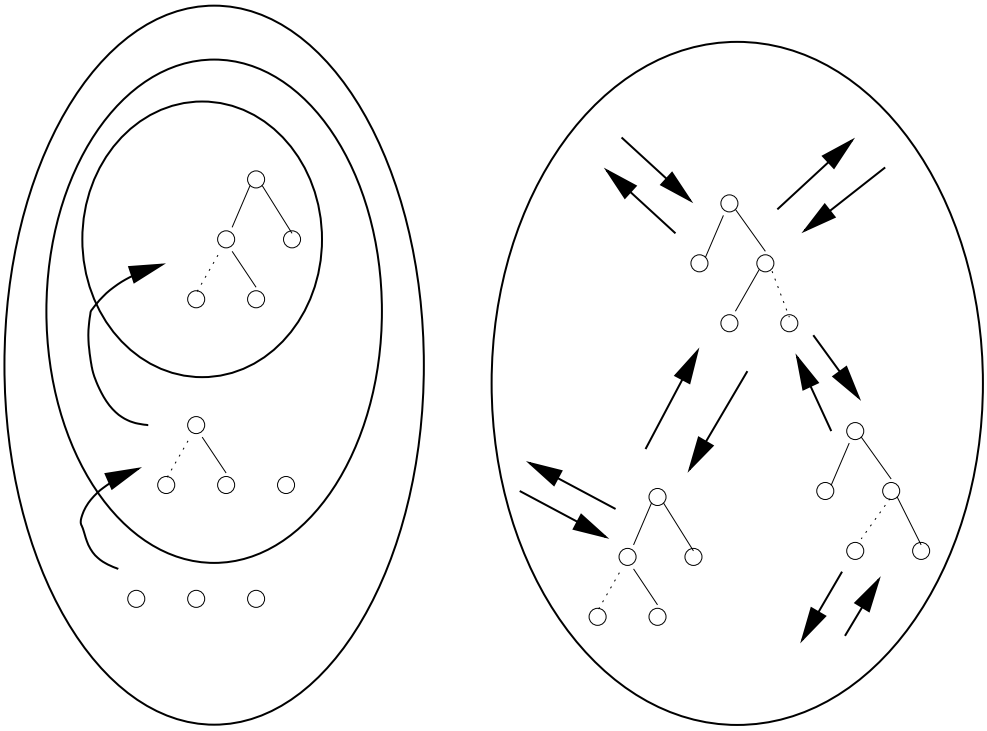
1.7 Focus

In this book, we consider only the compilation of queries. We leave out many special aspects like query optimization for multi-media database systems or multidatabase systems. These are just two omissions. We further do not consider the translation of update statements which — especially in the presence of triggers — can become quite complex. Furthermore, we assume the reader to be familiar with the fundamentals of database systems [252, 472, 624, 680, 786] and their implementation [397, 300]. Especially, knowledge on query execution engines is required [338].

Last, the book presents a very personal view on query optimization. To see other views on the same topic, I strongly recommend to read the literature cited in this book and the references found therein. A good start are overview articles, PhD theses, and books, e.g. [865, 313, 432, 433, 456, 527] [593, 597, 636, 799, 821, 848, 849].

1.8 Organization of the Book

The first part of the book is an introduction to the topic. It should give an idea about the breadth and depth of query optimization. We first recapitulate query optimization the way it is described in numerous text books on database systems. There should be nothing really new here except for some pitfalls we will point out. The Chapter 3 is devoted to the join ordering problem. This has several reasons. First of all, at least



a) Generative Approach b) Transformational Approach

Figure 1.8: Generation vs. transformation

one of the algorithms presented in this chapter forms the core of every plan generator. The second reason is that this problem allows to discuss some issues like search space sizes and problem complexities. The third reason is that we do not have to delve into details. We can stick to very simple (you might call them unrealistic) cost functions, do not have to concern ourselves with details of the runtime system and the like. Expressed positively, we can concentrate on some algorithmic aspects of the problem. In Chapter 4 we do the opposite. The reader will not find any advanced algorithms in this chapter but plenty of details on disks and cost functions. The goal of the rest of the book is then to bring these issues together, broaden the scope of the chapters, and treat problems not even touched by them. The main issue not touched is query rewrite.

Chapter 2

Textbook Query Optimization

Almost every introductory textbook on database systems contains a section on query optimization (or at least query processing) [252, 472, 624, 680, 786]. Also, the two existing books on implementing database systems contain a section on query optimization [397, 300]. In this chapter we give an excerpt[1] of these sections and subsequently discuss the problems with the described approach. The bottom line will be that these descriptions of query optimization capture the essence of it but contain pitfalls that need to be pointed out and gaps to be filled.

2.1 Example Query and Outline

We use the following relations for our example query:

Student(SNo, SName, SAge, SYear)
Attend(ASNo, ALNo, AGrade)
Lecture(LNo, LTitle, LPNo)
Professor(PNo, PName)

Those attributes belonging to the key of the relations have been underlined.

With the following query we ask for all students attending a lecture by a Professor called "*Larson*".

```
select  distinct s.SName
from    Student s, Attend a, Lecture l, Professor p
where   s.SNo = a.ASNo and a.ALNo = l.LNo
        and l.LPNo = p.PNo and p.PName = 'Larson'
```

The outline of the rest of the chapter is as follows. A query is typically translated into an algebraic expression. Hence, we first review the relational algebra and then discuss the translation process. Thereafter, we present the two phases of textbook query optimization: logical and physical query optimization. A brief discussion follows.

[1] We do not claim to be fair to the above mentioned sections.

2.2 Algebra

Let us briefly recall the standard definition of the most important algebraic operators. Their inputs are relations, that is sets of tuples. Sets do not contain duplicates. The attributes of the tuples are assumed to be simple (non-decomposable) values. The most common algebraic operators are defined in Fig. 2.1. Although the common set operations union ($\cup$), intersection ($\cap$), and setdifference ($\setminus$) belong to the relational algebra, we did not list them. Remember that $\cup$ and $\cap$ are both commutative and associative. $\setminus$ is neither of them. Further, for $\cup$ and $\cap$, two distributivity laws hold. However, since these operations are not used in this section, we refer to Figure 6.1 in Section 6.1.1.

Before we can understand Figure 2.1, we must clarify some terms and notations. For us, a *tuple* is a mapping from a set of attribute names (or attributes for short) to their corresponding values. These values are taken from certain domains. An actual tuple is denoted embraced by brackets. They include a comma-separated list of the form attribute name, column and attribute value as in `[name: ''Anton'', age: 2]`. If we have two tuples with different attribute names, they can be concatenated, i.e. we can take the union of their attributes. *Tuple concatentation* is denoted by '$\circ$'. For example `[name: ''Anton'', age: 2]` $\circ$ `[toy: ''digger'']` results in `[name: ''Anton'', age: 2, toy: ''digger'']`. Let A and A' be two sets of attributes where $A' \subseteq A$ holds. Further let t a tuple with schema A. Then, we can project t on the attributes in A (written as $t.A$). The resulting tuple contains only the attributes in A'; others are discarded. For example, if t is the tuple `[name: ''Anton'', age: 2, toy: ''digger'']` and $A = \{\texttt{name}, \texttt{age}\}$, then $t.A$ is the tuple `[name: ''Anton'', age: 2]`.

A relation is a set of tuples with the same attributes. The schema of a relation is the set of attributes. For a relation R this is sometimes denoted by `sch(`R`)`, the *schema* of R. We denote it by $\mathcal{A}(R)$ and extend it to any algebraic expression producing a set of tuples. That is, $\mathcal{A}(e)$ for any algebraic expression is the set of attributes the resulting relation defines. Consider the predicate `age` $= 2$ where `age` is an attribute name. Then, `age` behaves like a free variable that must be bound to some value before the predicate can be evaluated. This motivates us to often use the terms *attribute* and *variable* synonymously. In the above predicate, we would call `age` a free variable. The set of free variables of an expression e is denoted by $\mathcal{F}(e)$.

Sometimes it is useful to work with sequences of attributes in comparison predicates. Let $A = \langle a_1, \ldots, a_k \rangle$ and $B = \langle b_1, \ldots, b_k \rangle$ be two attribute sequences. Then for any comparison operator $\theta \in \{=, \leq, <, \geq, >, \neq\}$, the expression $A\theta B$ abbreviates $a_1 \theta b_1 \wedge a_2 \theta b_2 \wedge \ldots \wedge a_k \theta b_k$.

Often, a *natural join* is defined. Consider two relations R_1 and R_2. Define $A_i := \mathcal{A}(R_i)$ for $i \in \{1, 2\}$, and $A := A_1 \cap A_2$. Assume that A is non-empty and $A = \langle a_1, \ldots, a_n \rangle$. If A is non-empty, the natural join is defined as

$$R_1 \bowtie R_2 \quad := \quad \Pi_{A_1 \cup A_2}(R_1 \bowtie_p \rho_{A:A'}(R_2))$$

where $\rho_{A:A'}$ renames the attributes a_i in A to a'_i in A' and the predicate p has the form $A = A'$, i.e. $a_1 = a'_1 \wedge \ldots \wedge a_n = a'_n$.

For our algebraic operators, several equivalences hold. They are given in Figure 2.2. For them to hold, we typically require that the relations involved have disjoint

$$\begin{aligned}
\sigma_p(R) &:= \{r | r \in R, p(r)\} \\
\Pi_A(R) &:= \{r.A | r \in R\} \\
R_1 \times R_2 &:= \{r_1 \circ r_2 | r_1 \in R_1, r_2 \in R_2\} \\
R_1 \bowtie_p R_2 &:= \sigma_p(R_1 \times R_2)
\end{aligned}$$

Figure 2.1: Relational algebra

attribute sets. That is, we assume—even for the rest of the book—that attribute names are unique. This is often achieved by using the notation $R.a$ for a relation R or $v.a$ for a variable ranging over tuples with an attribute a. Another possibility is to use the renaming operator ρ.

Some equivalences are not always valid. Their validity depends on whether some condition(s) are satisfied or not. For example, Eqv. 2.4 requires $\mathcal{F}(p) \subseteq A$. That is, all attribute names occurring in p must be contained in the attribute set A the projection retains: otherwise, we could not evaluate p after the projection has been applied. Although all conditions in Fig. 2.2 are of this flavor, we will see throughout the course of the book that more complex conditions exist.

2.3 Canonical Translation

The next question is how to translate a given SQL query into the algebra. Since the relational algebra works on sets and not bags (multisets), we can only translate SQL queries that contain a **distinct**. Further, we restrict ourselves to flat queries not containing any subquery. Since negation, disjunction, aggregation, and quantifiers pose further problems, we neglect them. Further, we do not allow **group by**, **order by**, **union**, **intersection**, and **except** in our query. Last, we allow only attributes in the **select** clause and not more complex expressions. EX

Thus, the generic SQL query pattern we can translate into the algebra looks as follows:

select distinct $a_1, a_2, \ldots, a_m$
from $R_1 c_1, R_2 c_2, \ldots, R_n c_n$
where p

Here, the R_i are relation names and the c_i are correlation names. The a_i in the **select** clause are attribute names (or expressions of the form $c_i.a_i$) taken from the relations in the **from** clause. The predicate p is assumed to be a conjunction of comparisions between attributes or attributes and constants.

The translation process then follows the procedure described in Figure 2.3. First, we construct an expression that produces the cross product of the entries found in the **from** clause. The result is

$$((\ldots((R_1 \times R_2) \times R_3)\ldots) \times R_n).$$

$$\sigma_{p_1 \wedge \ldots \wedge p_k}(R) \equiv \sigma_{p_1}(\ldots(\sigma_{p_k}(R))\ldots) \quad (2.1)$$

$$\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R)) \quad (2.2)$$

$$\Pi_{A_1}(\Pi_{A_2}(\ldots(\Pi_{A_k}(R))\ldots)) \equiv \Pi_{A_1}(R) \text{ if } A_i \subseteq A_j \text{ for } i < j \quad (2.3)$$

$$\Pi_A(\sigma_p(R)) \equiv \sigma_p(\Pi_A(R)) \text{ if } \mathcal{F}(p) \subseteq A \quad (2.4)$$

$$(R_1 \times R_2) \times R_3 \equiv R_1 \times (R_2 \times R_3) \quad (2.5)$$

$$(R_1 \bowtie_{p_{1,2}} R_2) \bowtie_{p_{2,3}} R_3 \equiv R_1 \bowtie_{p_{1,2}} (R_2 \bowtie_{p_{2,3}} R_3) \text{ if } \mathcal{F}(p_{1,2}) \subseteq \mathcal{A}(R_1) \cup \mathcal{A}(R_2) \text{ and } \mathcal{F}(p_{2,3}) \subseteq \mathcal{A}(R_2) \cup \mathcal{A}(R_3) \quad (2.6)$$

$$R_1 \times R_2 \equiv R_1 \times R_2 \quad (2.7)$$

$$R_1 \bowtie_p R_2 \equiv R_2 \bowtie_p R_1 \quad (2.8)$$

$$\sigma_p(R_1 \times R_2) \equiv \sigma_p(R_1) \times R_2 \text{ if } \mathcal{F}(p) \subseteq \mathcal{A}(R_1) \quad (2.9)$$

$$\sigma_p(R_1 \bowtie_q R_2) \equiv \sigma_p(R_1) \bowtie_q R_2 \text{ if } \mathcal{F}(p) \subseteq \mathcal{A}(R_1) \quad (2.10)$$

$$\Pi_A(R_1 \times R_2) \equiv \Pi_{A_1}(R_1) \times \Pi_{A_2}(R_2) \text{ if } A = A_1 \cup A_2,\ A_i \subseteq \mathcal{A}(R_i) \quad (2.11)$$

$$\Pi_A(R_1 \bowtie_p R_2) \equiv \Pi_{A_1}(R_1) \bowtie_p \Pi_{A_2}(R_2) \text{ if } \mathcal{F}(p) \subseteq A,\ A = A_1 \cup A_2, \text{ and } A_i \subseteq \mathcal{A}(R_i) \quad (2.12)$$

$$\sigma_p(R_1 \theta R_2) \equiv \sigma_p(R_1) \theta \sigma_p(R_2) \text{ where } \theta \text{ is any of } \cup, \cap, \setminus \quad (2.13)$$

$$\Pi_A(R_1 \cup R_2) \equiv \Pi_A(R_1) \cup \Pi_A(R_2) \quad (2.14)$$

$$\sigma_p(R_1 \times R_2) \equiv R_1 \bowtie_p R_2 \quad (2.15)$$

Figure 2.2: Equivalences for the relational algebra

Next, we add a selection with the **where** predicate:

$$\sigma_p((\ldots((R_1 \times R_2) \times R_3)\ldots) \times R_n).$$

Last, we project on the attributes found in the **select** clause.

$$\Pi_{a_1,\ldots,a_n}(\sigma_p((\ldots((R_1 \times R_2) \times R_3)\ldots) \times R_n)).$$

For our example query

select **distinct** s.SName
from Student s, Attend a, Lecture l, Professor p
where s.SNo = a.ASNo **and** a.ALNo = l.LNo
and l.LPNo = p.PNo **and** p.PName = ‘Larson’

1. Let $R_1 \ldots R_k$ be the entries in the **from** clause of the query. Construct the expression

$$F := \begin{cases} R_1 & \text{if } k = 1 \\ ((\ldots(R_1 \times R_2) \times \ldots) \times R_k) & \text{else} \end{cases}$$

2. The **where** clause is optional in SQL. Therefore, we distinguish the cases that there is no **where** clause and that the **where** clause exists and contains a predicate p. Construct the expression

$$W := \begin{cases} F & \text{if there is no } \textbf{where} \text{ clause} \\ \sigma_p(F) & \text{if the } \textbf{where} \text{ clause contains } p \end{cases}$$

3. Let s be the content of the **select distinct** clause. For the canonical translation it must be of either '*' or a list $a_1, \ldots, a_n$ of attribute names. Construct the expression

$$S := \begin{cases} W & \text{if } s = \text{'*'} \\ \Pi_{a_1,\ldots,a_n}(W) & \text{if } s = a_1, \ldots, a_n \end{cases}$$

4. Return S.

Figure 2.3: (Simplified) Canonical translation of SQL to algebra

the result of the translation is

$$\Pi_{\text{s}.SName}(\sigma_p(((\text{S}tudent[s] \times \text{A}ttend[a]) \times \text{L}ecture[l]) \times \text{P}rofessor[p]))$$

where p equals

s.SNo = a.ASNo **and** a.ALNo = l.LNo **and** l.LPNo = p.PNo **and** p.PName = 'Larson'.

Note that we used the notation $R[r]$ to say that a relation named R has the correlation name r. During the course of the book we will be more precise about the semantics of this notation and it will deviate from the one suggested here. We will take r as a variable successively bound to the elements (tuples) in R. However, for the purpose of this chapter it is sufficient to think of it as associating a correlation name with a relation. The query is represented graphically in Figure 2.7 (top).

2.4 Logical Query Optimization

Textbook query optimization takes place in two separate phases. The first phase is called *logical query optimization* and the second *physical query optimization*. Figure 2.4 lists all these steps together with the translation step. In this section we discuss

1. translate query into its canonical algebraic expression
2. perform logical query optimization
3. perform physical query optimization

Figure 2.4: Text book query optimization

logical query optimization. The foundation for this step is formed by the set of algebraic equivalences (see Figure 2.2). The set of algebraic equivalences spans the potential search space for this step. Given an initial algebraic expression—resulting from the translation of the given query—the algebraic equivalences can be used to derive all algebraic expressions that are equivalent to the initial algebraic expression. This set of all equivalent algebraic expressions can be derived by applying the equivalences first to the initial expression and then to all derived expressions until no new expression is derivable. Thereby, the algebraic equivalences can be applied in both directions: from left to right and from right to left. Care has to be taken that the conditions attached to the equivalences are obeyed.

Of course, whenever we find a new algebraic equivalence that could not be derived from those already known, adding this equivalence increases our potential search space. On the one hand, this has the advantage that in a larger search space we may find better plans. On the other hand, it increases the already large search space which might cause problems for its exploration. Nevertheless, finding new equivalences is a well-established sport among database researchers.

One remark on *better* plans. Plans can only be compared if costs can be attached to them via some cost function. This is what happens in most industrial strength query optimizers. However, at the level of logical algebraic expressions adding precise costs is not possible: too many implementation details are missing. These are added to the plan during the next phase called *physical query optimization*. As a consequence, we are left with plans without costs. The only thing we can do is to *heuristically* judge the effectiveness of applying an equivalence from left to right or in the opposite direction. As always with heuristics, the hope is that they work for most queries. However, it is typically very easy to find counter examples where the heuristics do not result in the best plan possible. (Again, *best* with respect to some metrics.) This finding can be generalized: any query optimization that takes place in more than a single phase risks missing the best plan. This is an important observation and we will come back to this issue more than once.

After these words of warning let us continue to discuss textbook query optimization. Logical query optimization requires the organization of all equivalences into groups. Further, the equivalences are directed. That is, it is fixed whether they are applied in a left to right or right to left manner. A *directed* equivalence is called *rewrite rule*. The groups of rewrite rules are then successively applied to the initial algebraic expression. Figure 2.5 describes the different steps performed during logical query optimization. Associated with each step is a set of rewrite rules that are applied to the input expression to yield a result expression. The numbers correspond to the equiva-

1. break up conjunctive selection predicates
   (Eqv. 2.1: $\rightarrow$)
2. push down selections
   (Eqv. 2.2: $\rightarrow$), (Eqv. 2.9: $\rightarrow$)
3. introduce joins
   (Eqv. 2.15: $\rightarrow$)
4. determine join order
   Eqv. 2.8, Eqv. 2.6, Eqv. 2.5, Eqv. 2.7
5. introduce and push down projections
   (Eqv. 2.3: $\leftarrow$), (Eqv. 2.4: $\rightarrow$),
   (Eqv. 2.11: $\rightarrow$), (Eqv. 2.12: $\rightarrow$)

Figure 2.5: Logical query optimization

lences in Figure 2.2. A small arrow indicates the direction in which the equivalences are applied.

The first step breaks up conjunctive selection predicates. The motivation behind this step is that selections with simple predicates can be moved around easier. The rewrite rule used in this step is Equivalence 2.1 applied from left to right. For our example query Step 1 results in

$$\begin{array}{l}\Pi_{\text{s}.SName}(\\ \quad \sigma_{\text{s}.SNo=a.ASNo}(\\ \quad\quad \sigma_{\text{a}.ALNo=l.LNo}(\\ \quad\quad\quad \sigma_{\text{l}.LPNo=p.PNo}(\\ \quad\quad\quad\quad \sigma_{\text{p}.PName=\text{`}Larson'}(\\ \quad\quad\quad\quad\quad ((\text{S}tudent[s] \times \text{A}ttend[a]) \times \text{L}ecture[l]) \times \text{P}rofessor[p])))))\end{array}$$

The query is represented graphically in Figure 2.7 (middle).

Step 2 pushes selections down the operator tree. The motivation here is to reduce the number of tuples as early as possible such that subsequent (expensive) operators have smaller input. Applying this step to our example query yields:

$$\begin{array}{l}\Pi_{\text{s}.SName}(\\ \quad \sigma_{\text{l}.LPNo=p.PNo}(\\ \quad\quad \sigma_{\text{a}.ALNo=l.LNo}(\\ \quad\quad\quad \sigma_{\text{s}.SNo=a.ASNo}(\text{S}tudent[s] \times \text{A}ttend[a])\\ \quad\quad\quad \times \text{L}ecture[l])\\ \quad\quad \times(\sigma_{\text{p}.PName=\text{`}Larson'}(\text{P}rofessor[p]))))\end{array}$$

The query is represented graphically in Figure 2.7 (bottom).

Excursion In general, we might encounter problems when pushing down selections. It may be the case that the order of the cross products is not well-suited for pushing selections down. If this is the case, we must consider reordering cross products during this step (Eqv. 2.7 and 2.5). To illustrate this point consider the following example query.

select **distinct** s.SName
from Student s, Lecture l, Attend a
where s.SNo = a.ASNo **and** a.ALNo = l.LNo
and l.LTitle = 'Databases I'

After translation and Steps 1 and 2 the algebraic expression looks like

$$\Pi_{\text{s}.SName}(\sigma_{\text{s}.SNo=a.ASNo}(\sigma_{\text{a}.ALNo=l.LNo}(\text{S}tudent[s] \times \sigma_{\text{l}.LTitle=\text{'}Databases\ I\text{'}}(\text{L}ecture[l])) \times \text{A}ttend[a]))).$$

Neither of $\sigma_{\text{s}.SNo=a.ASNo}$ and $\sigma_{\text{a}.ALNo=l.LNo}$ can be pushed down further. Only after reordering the cross products such as in

$$\Pi_{\text{s}.SName}(\sigma_{\text{s}.SNo=a.ASNo}(\sigma_{\text{a}.ALNo=l.LNo}(\text{S}tudent[s] \times \text{A}ttend[a]) \times \sigma_{\text{l}.LTitle=\text{'}Databases\ I\text{'}}(\text{L}ecture[l]))))$$

can $\sigma_{\text{s}.SNo=a.ASNo}$ be pushed down:

$$\Pi_{\text{s}.SName}(\sigma_{\text{a}.ALNo=l.LNo}(\sigma_{\text{s}.SNo=a.ASNo}(\text{S}tudent[s] \times \text{A}ttend[a]) \times \sigma_{\text{l}.LTitle=\text{'}Databases\ I\text{'}}(\text{L}ecture[l]))))$$

This is the reason why in some textbooks reorder cross products before selections are pushed down [252]. In this appoach, reordering of cross products takes into account the selection predicates that can possibly be pushed down to the leaves and down to just prior a cross product. In any case, the Steps 2 and 4 are highly interdependent and there is no simple solution. □

After this small excursion let us resume rewriting our main example query. The next step to be applied is converting cross products to join operations (Step 3). The motivation behind this step is that the evaluation of cross products is very expensive and results in huge intermediate results. For every tuple in the left input an output tuple must be produced for every tuple in the right input. A join operation can be implemented much more efficiently. Applying Equivalence 2.15 from left to right to our example query results in

$$\Pi_{\text{s}.SName}((\text{S}tudent[s] \bowtie_{\text{s}.SNo=a.ASNo} \text{A}ttend[a]) \bowtie_{\text{a}.ALNo=l.LNo} \text{L}ecture[l]) \bowtie_{\text{l}.LPNo=p.PNo} (\sigma_{\text{p}.PName=\text{'}Larson\text{'}}(\text{P}rofessor[p])))$$

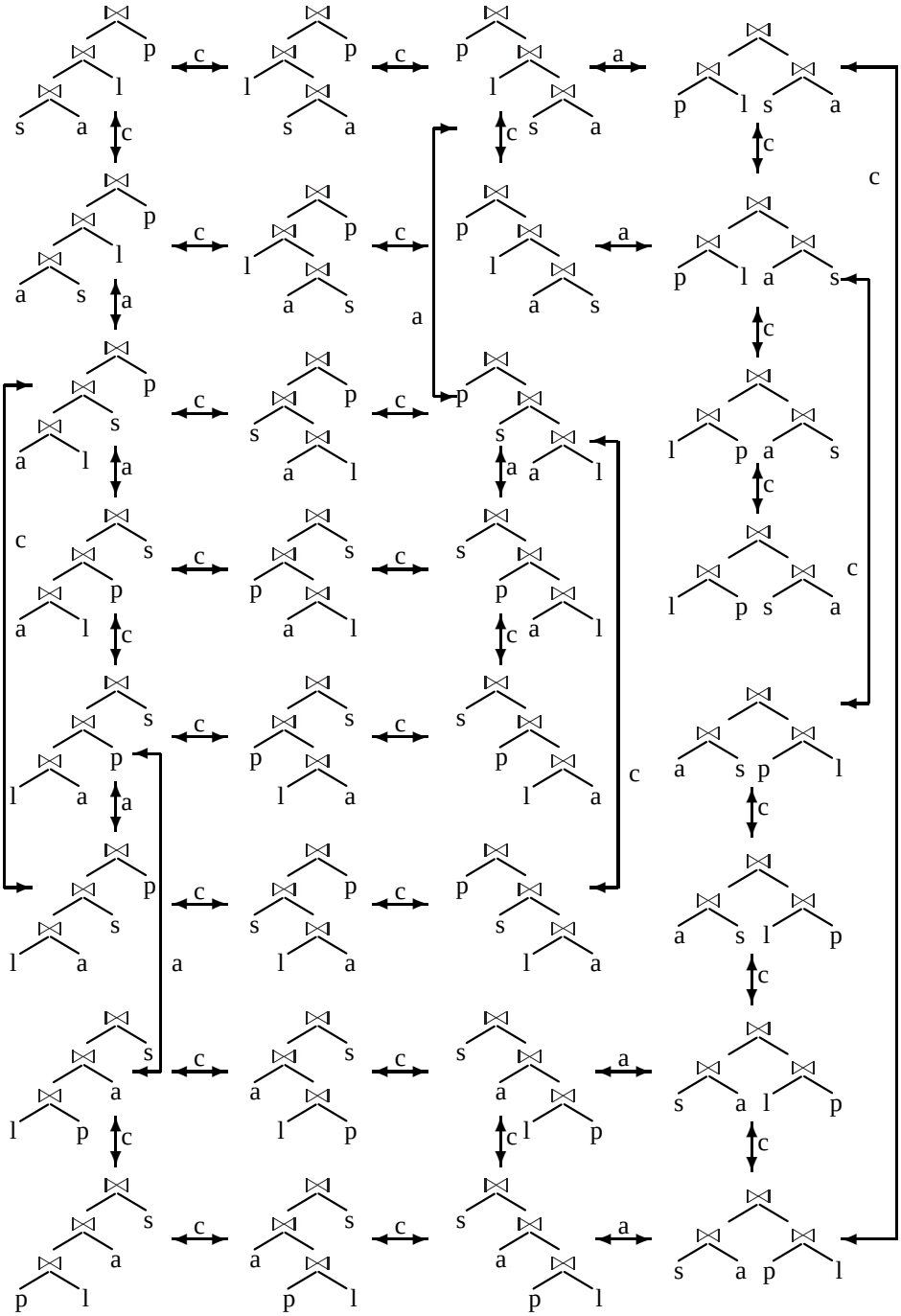


Figure 2.6: Different join trees

The query is represented graphically in Figure 2.8 (top).

The next step is really tricky and involved: we have to find an optimal order for evaluating the joins. The join's associativity and commutativity gives us plenty of alternative (equivalent) evaluation plans. For our rather simple query Figure 2.6 lists some of the possible join orders where we left out the join predicates and used the

single letter correlation names to denote the relations to be joined. Only p abbreviates the more complex expression $\sigma_{\text{p}.PName=\text{`}Larson\text{'}}(\text{P}rofessor[p])$. The edges show how plans can be derived from other plans by applying commutativity (c) or associativity (a).

Unfortunately, we cannot ignore the problem of finding a good join order. It has been shown that the order in which joins are evaluated has an enormous influence on the total evaluation cost of a query. Thus, it is an important problem. On the other hand, the problem is really tough. Most join ordering problems turn out to be NP-hard. As a consequence, many different heuristics and cost-based algorithms have been invented. They are discussed in depth in Chapter 3. There we will also find examples showing how important (in terms of costs) the right choice of the join order is.

To continue with our example query, we use a very simple heuristics: among all possible joins select the one first that produces the smallest intermediate result. This can be motivated as follows. In our current algebraic expression, the first join to be executed is

$$\text{S}tudent[s] \bowtie_{\text{s}.SNo=a.ASNo} \text{A}ttend[a].$$

All students and their attendances to some lecture are considered. The result and hence the input to the next join will be very big. On the other hand, if there is only one professor named *Larson*, the output of $\sigma_{\text{p}.PName=\text{`}Larson\text{'}}(\text{P}rofessor[p])$ is a single tuple. Joining this single tuple with the relation `Lecture` results in an output containing one tuple for every lecture taught by *Larson*. For a large university, this will be a small subset of all lectures. Continuing this line, we get the following algebraic expression:

$$\begin{array}{l}\Pi_{\text{s}.SName}(\\ \quad((\sigma_{\text{p}.PName=\text{`}Larson\text{'}}(\text{P}rofessor[p]) \\ \qquad \bowtie_{\text{p}.PNo=l.LPNo} \text{L}ecture[l]) \\ \qquad\quad \bowtie_{\text{l}.LNo=a.ALNo} \text{A}ttend[a]) \\ \qquad\qquad \bowtie_{\text{a}.ASno=s.SNo} \text{S}tudent[s])\end{array}$$

The query is represented graphically in Figure 2.8 (middle).

The last step minimizes intermediate results by projecting out irrelevant attributes. An attribute is irrelevant, if it is not used further up the operator tree. When pushing down projections, we only apply them just before a pipeline breaker [338]. The reason is that for pipelined operators like selection, eliminating superfluous attributes does not gain much. The only pipeline breaker occurring in our plan is the join operator. Hence, before a join is applied, we project on the attributes that are further needed. The result is

EX

$$\begin{array}{l}\Pi_{\text{s}.SName}(\\ \quad\Pi_{a.ASNo}(\\ \qquad\Pi_{l.LNO}(\\ \qquad\quad\Pi_{p.PNo}(\sigma_{\text{p}.PName=\text{`}Larson\text{'}}(\text{P}rofessor[p])) \\ \qquad\quad\bowtie_{\text{p}.PNo=l.LPNo} \\ \qquad\quad\Pi_{\text{l}.LPno,l.LNo}(\text{L}ecture[l])) \\ \qquad\bowtie_{\text{l}.LNo=a.ALNo} \\ \qquad\Pi_{\text{a}.ALNo,a.ASNo}(\text{A}ttend[a])) \\ \quad\bowtie_{\text{a}.ASno=s.SNo} \\ \quad\Pi_{\text{s}.SNo,s.SName}(\text{S}tudent[s]))\end{array}$$

This expression is represented graphically in Figure 2.8 (bottom).

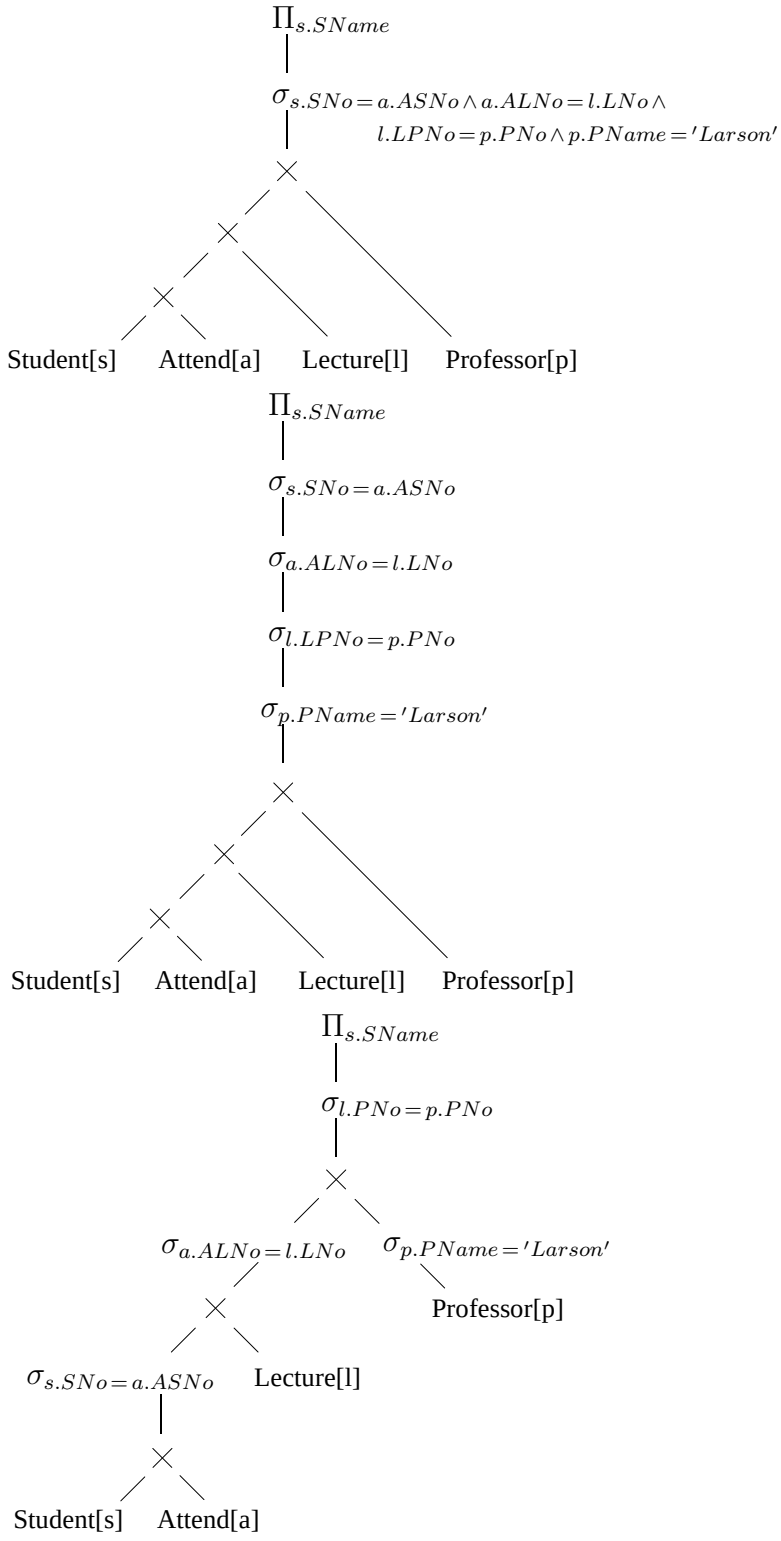


Figure 2.7: Plans for example query (Part I)

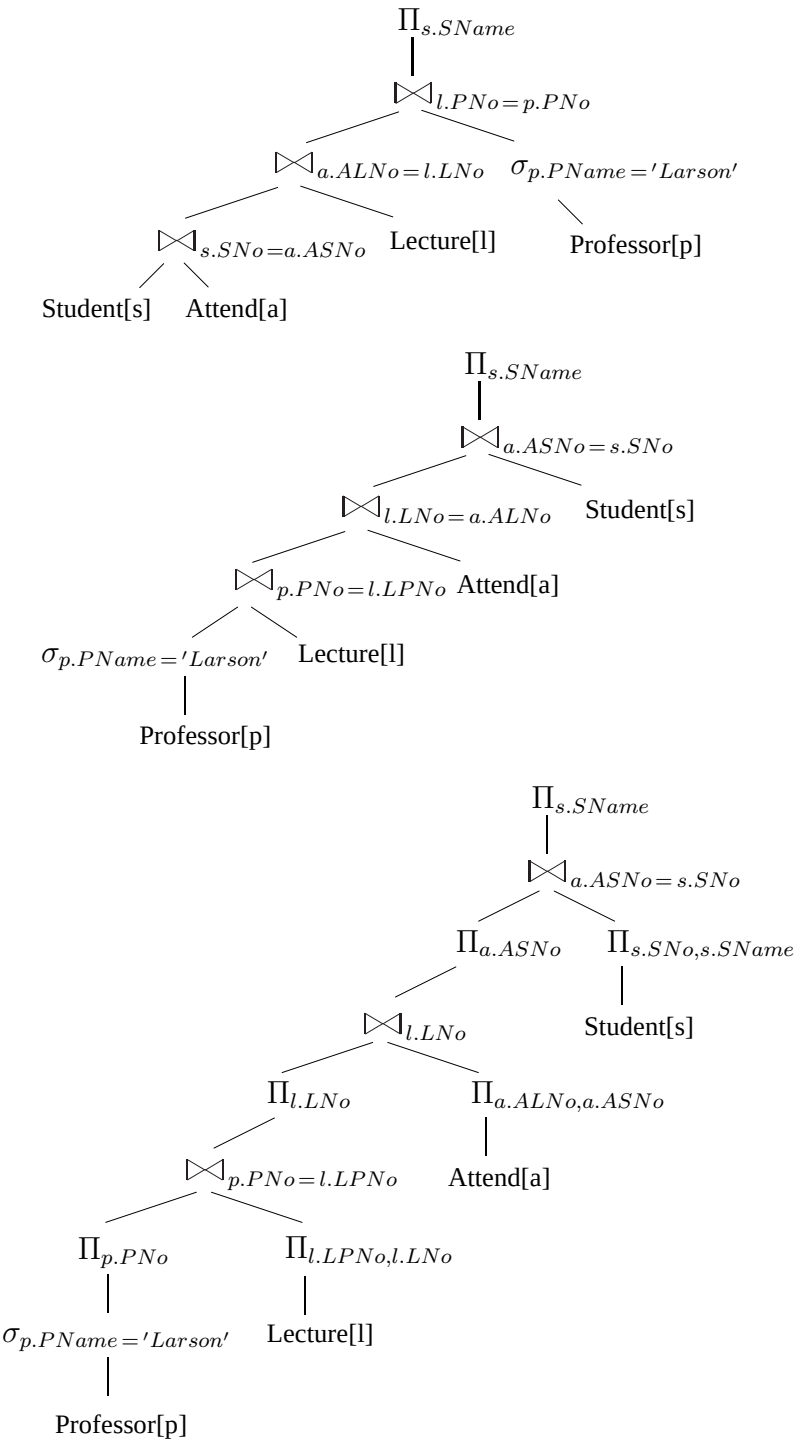


Figure 2.8: Plans for example query (Part II)

2.5 Physical Query Optimization

Physical query optimization adds more information to the logical query evaluation plan. First, there exist many different ways to access the data stored in a database. One

possibility is to scan a relation to find the relevant tuples. Another alternative is to use an index to access only the relevant parts. If an unclustered index is used, it might be beneficial to sort the *tuple identifiers* (TIDs[2]) to turn otherwise random disk accesses into sequential accesses. Since there is a multitutude of possibilities to access data, this topic is discussed in depth in Chapter 4. Second, the algebraic operators used in the logical plan may have different alternative implementations. The most prominent example is the join operator that has many different implementations: simple nested loop join, blockwise nested loop join, blockwise nested loop join with in-memory hash table, index nested loop join, hybrid hash join, sort merge join, bandwidth join, special spatial joins, set joins, and structural joins. Most of these join implementations can be applied only in certain situations. Most algorithms only implement equi-joins where the join predicate is a conjunction of simple equalities. Further, all the implementations differ in cost and robustness. But also other operators like grouping may have alternative implementations. Typically, for these operators exist sort-based and hash-based alternatives. Third, some operators require certain *properties* for their input streams. For example, a sort merge join requires its input to be sorted on the join attributes occurring in the equalities of the join predicate. These attributes are called *join attributes*. The sortedness property can be *enforced* by a sort operator. The sort operator is thus an *enforcer* since it makes sure that the required property holds. As we will see, properties and enforcers play a crucial role during plan generation.

If common subexpressions are detected at the algebraic level, it might be beneficial to compute them only once and store the result. To do so, a *tmp* operator must be introduced. Later on, we will see more of these operators that materialize (partial) intermediate results in order to avoid the same computation to be performed more than once. An alternative is to allow QEPs which are DAGs and not merely trees (see Section **??**).

Physical query optimization is concerned with all the issues mentioned above. The outline of it is given in Figure 2.9. Let us demonstrate this for our small example query. Let us assume that there exists an index on the name of the professors. Then, instead of scanning the whole professor relation, it is beneficial to use the index to retrieve only those professors named *Larson*. Further, since a sort merge join is very robust and not the slowest alternative, we choose it as an implementation for all our join operations. This requires that we sort the inputs to the join operator on the join attributes. Since sorting is a pipeline breaker, we introduce it between the projections and the joins. The resulting plan is

$$
\begin{array}{l}
\Pi_{\text{s}.SName}(\\
\quad \text{Sort}_{\text{a}.ASNo}(\Pi_{\text{a}.ASNo}(\\
\quad\quad \text{Sort}_{\text{l}.LNo}(\Pi_{\text{l}.LNO}(\\
\quad\quad\quad \text{Sort}_{\text{p}.PNo}(\Pi_{\text{p}.PNo}(\text{IdxScan}_{\text{p}.PName=`Larson'}(\text{P}rofessor[p])))) \\
\quad\quad\quad \bowtie^{smj}_{\text{p}.PNo=l.LPNo} \\
\quad\quad\quad \text{Sort}_{\text{l}.LPNo}(\Pi_{\text{l}.LPno,l.LNo}(\text{L}ecture[l]))) \\
\quad\quad \bowtie^{smj}_{\text{l}.LNo=a.ALNo} \\
\quad\quad \text{Sort}_{\text{a}.ALNo}(\Pi_{\text{a}.ALNo,a.ASNo}(\text{A}ttend[a])))) \\
\quad \bowtie^{smj}_{\text{a}.ASno=s.SNo}
\end{array}
$$

[2]Sometimes TIDs are called RIDs (Row Identifiers).

1. introduce index accesses
2. choose implementations for algebraic operators
3. introduce physical operators (sort, tmp)

Figure 2.9: Physical query optimization

$$\text{Sort}_{\text{s}.SNo}(\Pi_{\text{s}.SNo,s.SName}(\text{S}Student[s]))))$$

where we annotated the joins with *smj* to indicate that they are sort merge joins. The *sort* operator has the attributes on which to sort as a subscript. We cheated a little bit with the notation of the index scan. The index is a physical entity stored in the database. An index scan typically allows to retrieve the TIDs of the tuples qualifying the predicate. If this is the case, another access to the relation itself is necessary to fetch the relevant attributes (p.PNo in our case) from the qualifying tuples of the relation. This issue is rectified in Chapter 4. The plan is shown as an operator graph in Figure 2.10.

2.6 Discussion

This chapter left open many interesting issues. We took it for granted that the presentation of a query is an algebraic expression or operator tree. Is this really true? We have been very vague about ordering joins and cross products. We only considered queries of the form **select distinct**. How can we assure correct duplicate treatment for **select all**? We separated query optimization into two distinct phases: logical and physical query optimization. Any separation into different phases results in the danger of not producing an optimal plan. Logical query optimization turned out to be a little difficult: pushing selections down and reordering joins are mutually interdependent. How can we integrate these steps into a single one and thereby avoid the problem mentioned? Further, our logical query optimization was not cost based and cannot be: too much information is still missing from the plan to associate precise costs with a logical algebraic expression. How can we integrate the phases? How can we determine the costs of a plan? We covered only a small fraction of SQL. We did not discuss disjunction, negation, union, intersection, except, aggregate functions, group-by, order-by, quantifiers, outer joins, and nested queries. Furthermore, how about other query languages like OQL, XPath, XQuery? Further, enhancements like materialized views exist nowadays in many commercial systems. How can we exploit them beneficially? Can we exploit semantic information? Is our exploitation of index structures complete? What happens if we encounter NULL-values? Many questions and open issues remain. The rest of the book is about filling these gaps.

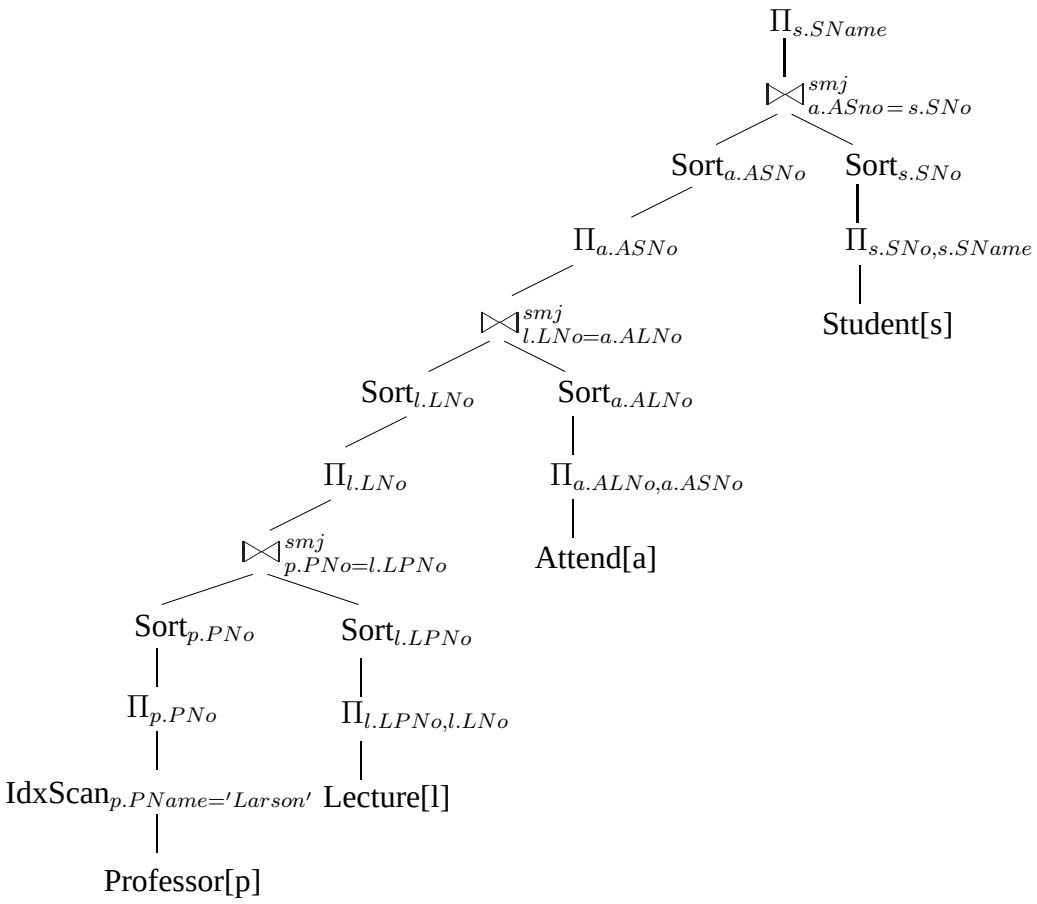


Figure 2.10: Plan for example query after physical query optimization

Chapter 3

Join Ordering

The problem of *join ordering* is a very restricted and — at the same time — a very complex one. We have touched this issue while discussing logical query optimization in Chapter 2. Join ordering is performed in Step 4 of Figure 2.5. In this chapter, we simplify the problem of join ordering by not considering duplicates, disjunctions, quantifiers, grouping, aggregation, or nested queries. Expressed positively, we concentrate on conjunctive queries with simple and cheap join predicates. What this exactly means will become clear in the next section. Subsequent sections discuss different algorithms for solving the join ordering problem. Finally, we take a look at the structure of the search space. This is important if different join ordering algorithms are compared via benchmarks. If the wrong parameters are chosen, benchmark results can be misleading.

The algorithms of this chapter form the core of every plan generator.

3.1 Queries Considered

A *conjunctive query* is one whose **where** clause contains a (complex) predicate which in turn is a conjunction of (simple) predicates. Hence, a conjunctive query involves only **and** and no *or* or *not* operations. A *simple predicate* is of the form $e_1 \theta e_2$ where $\theta \in \{=, \neq, <, >, \leq, \geq\}$ is a comparison operator and the e_i are simple expressions in attribute names possibly containing some simple and cheap arithmetic operators. By cheap we mean that it is not worth applying extra optimization techniques. In this chapter, we restrict simple predicates even further to the form $A = B$ for attributes A and B. A and B must also belong to different relations such that every simple predicate in this chapter is a join predicate. There are two reasons for this restriction. First, the most efficient join algorithms rely on the fact that the join predicate is of the form $A = B$. Such joins are called *equi-joins*. Any other join is called a *non-equi-join*. Second, in relational systems joins on foreign key attributes of one relation and key attributes of the other relation are very common. Other joins are rare.

A *base relation* is a relation that is stored (explicitly) in the database. For the rest of the chapter, let R_i ($1 \leq i \leq n$) be n relations. These relations can be base relations but do not necessarily have to be. They could also be base relations to which predicates have already been supplied, e.g. as a result of applying the first three steps of logical query optimization.

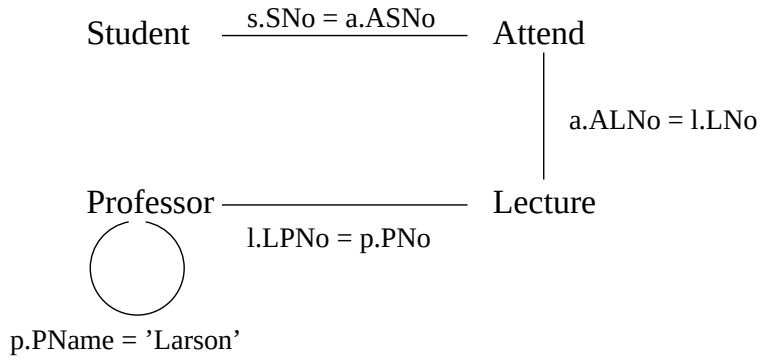


Figure 3.1: Query graph for example query of Section 2.1

Summarizing, the queries we consider can be expressed in SQL as

select distinct *
from $R_1,\ldots,R_n$
where p

where p is a conjunction of simple join predicates with attributes from exactly two relations. The latter restriction is not really necessary for the algorithms presented in this chapter but simplifies the exposition.

3.1.1 Query Graph

A query graph is a convenient representation of a query. It is an undirected graph with nodes $R_1, \ldots, R_n$. For every simple predicate in the conjunction P whose attributes belong to the relations R_i and R_j, we add an edge between R_i and R_j. This edge is labeled by the simple predicate. From now on, we denote the join predicate connecting R_i and R_j by $p_{i,j}$. In general, $p_{i,j}$ can be a conjunction of simple join predicates connecting R_i and R_j.

If query graphs are used for more than join ordering, selections need to be represented. This is done by self-edges from the relation to which the selection applies to itself. For the example query of Chapter 2.6, Figure 3.1 contains the according query graph.

Query graphs can have many different shapes. The shapes that play a certain role in query optimization and the evaluation of join ordering algorithms are shown in Fig. 3.2. The query graph classes relevant for this chapter are chain queries, star queries, tree queries, cyclic queries and clique queries. Note that these classes are not disjoint and
EX that some classes are subsets of other classes.

In this chaper, we only treat connected query graphs. These can be evaluated without cross products.

Excursion In general, the query graph is a hypergraph [850] as the following example shows.

```
select *
from R1, R2, R3, R4
```

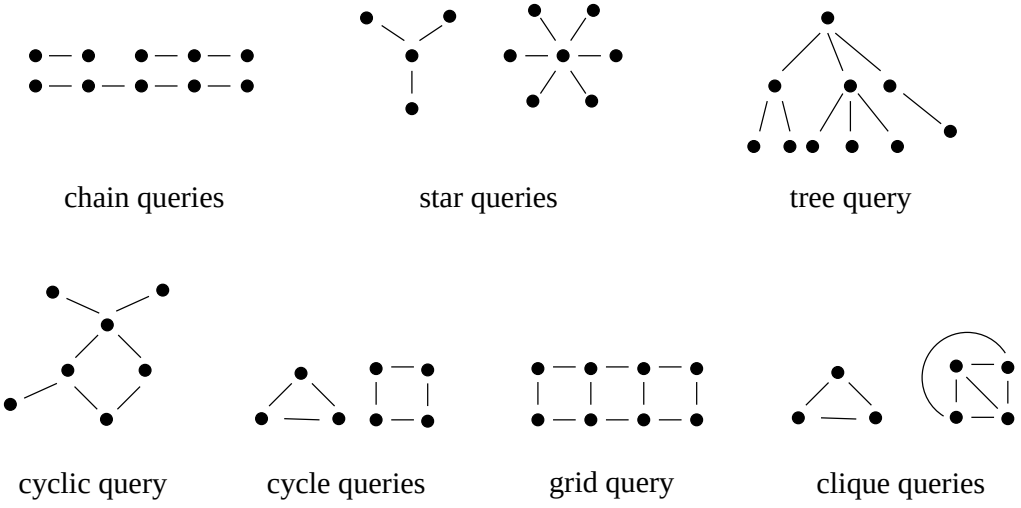


Figure 3.2: Query graph shapes

```
where f(R1.a, R2.a,R3.a) = g(R2.b,R3.b,R4.b)
```

3.1.2 Join Tree

A join tree is an algebraic expression in relation names and join operators. Sometimes, cross products are allowed, too. A cross product is the same as a join operator with *true* as its join predicate. A join tree has its name from its graph representation. There, a join tree is a binary tree whose leaf nodes are the relations and whose inner nodes are joins (and possibly cross products). The edges represent the input/output relationship. Examples of join trees have been shown in Figure 2.6.

Join trees fall into different classes. The most important classes are left-deep trees, right-deep trees, zig-zag trees, and bushy trees. *Left-deep trees* are join trees where every join has one of the relations R_i as its right input. *Right-deep trees* are defined analogously. In *zig-zag trees* at least one input of every join is a relation R_i. The class of zig-zag trees contains both left-deep and right-deep trees. For *bushy trees* no restriction applies. Hence, the class of bushy trees contains all of the above three classes. The roots of these notions date back to the paper by Selinger et al. [750], where the search space of the query optimizer was restricted to left-deep trees. There are two main reasons for this restriction. First, only one intermediate result is generated at any time during query evaluation. Second, the number of left-deep trees is far less than the number of e.g. bushy trees. The other classes were then added later by other researchers whenever they found better join trees in them. The different classes are illustrated in Figure 2.6. From left to right, the columns contain left-deep, zig-zag, right-deep, and bushy trees.

Left-deep trees directly correspond to an ordering (i.e. a permutation) of the relations. For example, the left-deep tree

$$((((R_2 \bowtie R_3) \bowtie R_1) \bowtie R_4) \bowtie R_5)$$

directly corresponds to the permutation R_2, R_3, R_1, R_4, R_5. It should be clear that there is a one-to-one correspondence between permutations and left-deep join trees.

We will also use the term *sequence of relations* synonymously. The notion of *join ordering* goes back to the times where only left-deep trees were considered and, hence, producing an optimal join tree was equivalent to optimally ordering the joins, i.e. determining a permutation with lowest cost.

Left-deep, right-deep, and zig-zag trees can be classed under the general term *linear trees*. Sometimes, the term linear trees is used synonymously for left-deep trees. We will not do so. Join trees are sometimes called *operator trees* or *query evaluation plans*. Although this is not totally wrong, these terms have a slightly different connotation. Operator trees typically contain more than only join operators. Query evaluation plans (QEPs or plans for short) typically have more information from physical query optimization associated with them.

3.1.3 Simple Cost Functions

In order to judge the quality of join trees, we need a cost function that associates a certain positive cost with each join tree. Then, the *task of join ordering* is to find among all equivalent join trees the join tree with lowest associated costs.

One part of any cost function are cardinality estimates. They are based on the cardinalities of the relations, i.e. the number of tuples contained in them. For a given relation R_i, we denote its cardinality by $|R_i|$.

Then, the cardinality of intermediate results must be estimated. This is done by introducing the notion of join selectivity. Let $p_{i,j}$ be a join predicate between relations R_i and R_j. The *selectivity* $f_{i,j}$ of $p_{i,j}$ is then defined as

$$f_{i,j} = \frac{|R_i \bowtie_{p_{i,j}} R_j|}{|R_i| * |R_j|}$$

This is the number of tuples in the join's result divided by the number of tuples in the Cartesian Product between R_i and R_j. If $f_{i,j}$ is 0.1, then only 10% of all tuples in the Cartesian Product survive the predicate $p_{i,j}$. Note that the selectivity is always a number between 0 and 1 and that $f_{i,j} = f_{j,i}$. We use an f and not an s, since the selectivity of a predicate is often called *filter factor*.

Besides the relation's cardinalities, the selectivities of the join predicates $p_{i,j}$ are assumed to be given as input to the join ordering algorithm. Therefore, we can compute the output cardinality of a join $R_i \bowtie_{p_{i,j}} R_j$, as

$$|R_i \bowtie_{p_{i,j}} R_j| = f_{i,j}|R_i||R_j|$$

From this it becomes clear that if there is no join predicate for two relations R_i and R_j, we can assume a join predicate *true* and associate a selectivity of 1 with it. The output cardinality is then the cardinality of the cross product between R_i and R_j. We also define $f_{i,i} = 1$ for all $1 \leq i \leq n$. This allows us to keep subsequent formulas simple.

We now need to extend our cardinality estimation to join trees. This can be done by recursively applying the above formula. Consider a join tree T joining two join trees T_1 and T_2, i.e. $T = T_1 \bowtie T_2$. Then, the result cardinality $|T|$ can be calculated as follows. If T is a leaf R_i, then $|T| := |R_i|$. Otherwise,

$$|T| = (\prod_{R_i \in R_1, R_j \in T_2} f_{i,j})\ |T_1|\ |T_2|.$$

Note that this formula assumes that the selectivities are independent of each other. Assuming independence is common but may be very misleading. More on this issue can be found in Chapter 27. Nevertheless, we assume independence and stick to the above formula.

For sequences of joins we can give a simple cardinality definition. Let $s = R_1, \ldots, R_n$ be a sequence of relations. Then

$$|s| = \prod_{k=1}^{n} (\prod_{i=1}^{k} f_{i,k} |R_k|).$$

Given the above, a query graph alone is not really sufficient for the specification of a join ordering problem: cardinalities and selectivities are missing. On the other hand, from a complete list of cardinalities and selectivities we can derive the query graph. Obviously, the following defines a chain query with query graph $R_1 - - - R_2 - - - R_3$:

$$\begin{aligned} |R_1| &= 10 \\ |R_2| &= 100 \\ |R_3| &= 1000 \\ f_{1,2} &= 0.1 \\ f_{2,3} &= 0.2 \end{aligned}$$

In all examples, we assume for all i and j for which $f_{i,j}$ is not given that there is no join predicate and hence $f_{i,j} = 1$.

We now come to cost functions. The first cost function we consider is called C_{out}. For a join tree T, $C_{\text{out}}(T)$ is the sum of all output cardinalities of all joins in T. Recursively, we can define C_{out} as

$$C_{\text{out}}(T) = \begin{cases} 0 & \text{if } T \text{ is a single relation} \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

From a theoretial point of view, C_{out} has many interesting properties: it is symmetric, it has the ASI property, and it can be applied to an expression of the logical algebra. From a practical point of view, however, it is rarely applied (yet).

In real cost functions, the cardinalities only serve as input to more complex formulas capturing the costs of a join implementation. Since real cost functions are too complex for this section, we stick to simple cost functions proposed by Krishnamurthy, Boral, and Zaniolo [503]. They argue that these cost functions are appropriate for main memory database systems. For the three different join implementations nested loop join (nlj), hash join (hj), and sort merge join (smj), they give the following cost functions:

$$\begin{aligned} C_{\text{nlj}}(e_1 \bowtie_p e_2) &= |e_1||e_2| \\ C_{\text{hj}}(e_1 \bowtie_p e_2) &= h|e_1| \\ C_{\text{smj}}(e_1 \bowtie_p e_2) &= |e_1|log(|e_1|) + |e_2|log(|e_2|) \end{aligned}$$

where e_i are join trees and h is the average length of the collision chain in the hash table. We will assume $h = 1.2$. All these cost functions are defined for a single join

operator. The cost of a join tree is defined as the sum of the costs of all joins it contains. We use the symbols C_x to also denote the costs of not only a single join but the costs of the whole tree. Hence, for sequences s of relations, we have

$$\begin{aligned} C_{\text{nlj}}(s) &= \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| * |s_i| \\ C_{\text{hj}}(s) &= \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}| \\ C_{\text{smj}}(s) &= \sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| \log(|s_1, \ldots, s_{i-1}|) + \sum_{i=1}^{n} |s_i| \log(|s_i|) \end{aligned}$$

Some notes on the cost functions are in order. First, note that these cost functions are even for main memory a little incomplete. For example, constant factors are missing. Second, the cost functions are mainly devised for left-deep trees. This becomes apparent when looking at the costs of hash joins. It is assumed that the right input is already stored in an appropriate hash table. Obviously, this can only hold for base relations, giving rise to left-deep trees. Third, C_{hj} and C_{smj} do not work for cross products. However, we can extend these cost functions by defining the cost of a cross product to be equal to its output cardinality, which happens to be the cost of C_{nlj}. Fourth, in reality, more complex cost functions are used and other parameters like the width of the tuples—i.e. the number of bytes needed to store them—also play an important role. Fifth, the above cost functions assume that the same join algorithm is chosen throughout the whole plan. In practice, this will not be true.

For the above chain query, we compute the costs of different join trees. The last join tree contains a cross product.

	C_{out}	C_{nlj}	C_{hj}	C_{smj}
$R_1 \bowtie R_2$	100	1000	12	697.61
$R_2 \bowtie R_3$	20000	100000	120	10630.26
$R_1 \times R_3$	10000	10000	10000	10000.00
$(R_1 \bowtie R_2) \bowtie R_3$	20100	101000	132	11327.86
$(R_2 \bowtie R_3) \bowtie R_1$	40000	300000	24120	32595.00
$(R_1 \times R_3) \bowtie R_2$	30000	1010000	22000	143542.00

For the calculation of C_{out} note that $|R_1 \bowtie R_2 \bowtie R_3| = 20000$ is included in all of the last three lines of its column. For the nested loop cost function, the costs are calculated as follows:

$$\begin{aligned} C_{\text{nlj}}((R_1 \bowtie R_2) \bowtie R_3) &= 1000 + 100 * 1000 = 101000 \\ C_{\text{nlj}}((R_2 \bowtie R_3) \bowtie R_1) &= 100000 + 20000 * 10 = 300000 \\ C_{\text{nlj}}((R_1 \times R_3) \bowtie R_2) &= 10000 + 10000 * 100 = 1010000 \end{aligned}$$

The reader should verify the other costs.

Several observations can be made from the above numbers:

- The costs of different join trees differ vastly under every cost function. Hence, it is worth spending some time to find a cheap join order.

- The costs of the same join tree differ under the different cost functions.
- The cheapest join tree is $(R_1 \bowtie R_2) \bowtie R_3$ under all four cost functions.
- Join trees with cross products are expensive.
  Thus, a heuristics often used is not to consider join trees that contain unnecessary cross products. (If the query graph consists of several unconnected components, then and only then cross products are necessary. In other words: if the query graph is connected, no cross products are necessary.).
- The join order matters even for join trees without cross products.

We would like to emphasize that the join order is also relevant under other cost functions.

Avoiding cross products is not always beneficial, as the following query specifiation shows:

$$\begin{aligned} |R_1| &= 1000 \\ |R_2| &= 2 \\ |R_3| &= 2 \\ f_{1,2} &= 0.1 \\ f_{1,3} &= 0.1 \end{aligned}$$

For C_{out} we have costs

Join Tree	C_{out}
$R_1 \bowtie R_2$	200
$R_2 \times R_3$	4
$R_1 \bowtie R_3$	200
$(R_1 \bowtie R_2) \bowtie R_3$	240
$(R_2 \times R_3) \bowtie R_1$	44
$(R_1 \bowtie R_3) \bowtie R_2$	240

Note that although the absolute numbers are quite small, the ratio of the best and the second best join tree is quite large. The reader is advised to find more examples and to apply other cost functions.

The following example illustrates that a bushy tree can be superior to any linear tree. Let us use the following query specification:

$$\begin{aligned} |R_1| &= 10 \\ |R_2| &= 20 \\ |R_3| &= 20 \\ |R_4| &= 10 \\ f_{1,2} &= 0.01 \\ f_{2,3} &= 0.5 \\ f_{3,4} &= 0.01 \end{aligned}$$

If we do not consider cross products, we have for the symmetric (see below) cost function C_{out} the following join trees and costs:

Join Tree	C_{out}
$R_1 \bowtie R_2$	2
$R_2 \bowtie R_3$	200
$R_3 \bowtie R_4$	2
$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$	24
$((R_2 \bowtie R_3) \bowtie R_1) \bowtie R_4$	222
$(R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4)$	6

Note that all other linear join trees fall into one of these classes, due to the symmetry of the cost function and the join ordering problem. Again, the reader is advised to find more examples and to apply other cost functions.

If we want to annotate a join operator by its implementation—which is necessary for the correct computation of costs—we write $\bowtie^{\text{impl}}$ for an implementation `impl`. For example, $\bowtie^{\text{smj}}$ is a sort-merge join, and the according cost function C_{smj} is used to compute its costs.

Two properties of cost functions have some impact on the join ordering problem. The first is symmetry. A cost function C_{impl} is called *symmetric* if $C_{\text{impl}}(R_1 \bowtie^{\text{impl}} R_2) = C_{\text{impl}}(R_2 \bowtie^{\text{impl}} R_1)$ for all relations R_1 and R_2. For symmetric cost functions, it does not make sense to consider commutativity. Hence, it suffices to consider left-deep trees only if we want to restrict ourselves to linear join trees. Note that C_{out}, C_{nlj}, C_{smj} are symmetric while C_{hj} is not.

The other property is the *adjacent sequence interchange* (ASI) property. Informally, the ASI property states that there exists a rank function such that the order of two subsequences is optimal if they are ordered according to the rank function. The ASI property is formally defined in Section 3.2.2. Only for tree queries and cost functions with the ASI property, a polynomial algorithm to find an optimal join order is known. Our cost functions C_{out} and C_{hj} have the ASI property, C_{smj} does not. Summarizing the properties of our cost functions, we see that the classification is orthogonal:

	ASI	$\neg$ ASI
symmetric	C_{out}, C_{nlj}	C_{smj}
$\neg$ symmetric	C_{hj}	(see text)

For the missing case of a non-symmetric cost function not having the ASI property, we can use the cost function of the hybrid hash join [231, 650].

We turn to another not really well-researched topic. The goal is to cut down the number of cost functions which have to be considered for optimization and to possibly allow for simpler cost functions, which saves time during plan generation. Unfortunately, we have to restrict ourselves to left-deep join trees. Let s denote a sequence or permutation of a given set of joins. We define an equivalence relation on cost functions.

Definition 3.1.1 *Let C and C' be two cost functions. Then*

$$C \equiv C' :\prec\succ (\forall s\ C(s) \textit{ minimal} \prec\succ C'(s) \textit{ minimal})$$

Here, s is a join sequence.

Obviously, $\equiv$ is an equivalence relation.

Now we can define the ΣIR property.

Definition 3.1.2 *A cost function C is* $\Sigma IR :\prec\succ C \equiv C_{out}$.

That is, ΣIR is the set of all cost functions that are equivalent to C_{out}.

Let us consider a very simple example. The last element of the sum in C_{out} is the size of the final join (all relations are joined). This is not the case for the following cost function:

$$C'_{\text{out}}(s) := \sum_{i=2}^{n-1} |s_1, \ldots, s_i|$$

Obviously, we have C'_{out} is ΣIR. The next observation shows that we can construct quite complex ΣIR cost functions:

Observation 3.1.3 *Let C_1 and C_2 be two ΣIR cost functions. For non-decreasing functions $f_1 : R \to R$ and $f_2 : R \times R \to R$ and constants $c \in R$ and $d \in R^+$, we have that* EX

$C_1 + c$
$C_1 * d$
$f_1 \circ C_1$
$f_2 \circ (C_1, C_2)$

are ΣIR. Here, $\circ$ denotes function composition and $(\cdot, \cdot)$ function pairing.

There are of course many more possibilites of constructing ΣIR functions. For the cost functions C_{hj}, C_{smj}, and C_{nlj}, we now investigate which of them have the ΣIR property.

Let us consider C_{hj} first. From

$$\begin{aligned} C_{\text{hj}}(s) &= \sum_{i=2}^{n} 1.2|s_1, \ldots, s_{i-1}| \\ &= 1.2|s_1| + 1.2 \sum_{i=2}^{n-1} |s_1, \ldots, s_i| \\ &= 1.2|s_1| + 1.2C'_{out}(s) \end{aligned}$$

and observation 3.1.3, we conclude that C_{hj} is ΣIR for a fixed relation to be joined first. If we can optimize C_{out} in polynomial time, then we can optimize C_{out} for a fixed starting relation. Indeed, by trying each relation as a starting relation, we can find the optimal join tree in polynomial time. An algorithm that computes the optimal solution for an arbitrary relation to be joined first can be found in Section 3.2.2. EX

Now, consider C_{smj}. Since

$$\sum_{i=2}^{n} |s_1, \ldots, s_{i-1}| log(|s_1, \ldots, s_{i-1}|)$$

is minimal if and only if

$$\sum_{i=2}^{n} |s_1, \ldots, s_{i-1}|$$

is minimal and $\sum_{i=2}^{n} |s_i| \log(|s_i|)$ is independent of the order of the relations within s — that is constant — we conclude that C_{smj} is ΣIR.

Last, we have that C_{nlj} is not ΣIR. To see this, consider the following counter example with three relations R_1, R_2, and R_3 of sizes 10, 10, and 100, resp. The selectivities are $f_{1,2} = \frac{9}{10}$, $f_{2,3} = \frac{1}{10}$, and $f_{1,3} = \frac{1}{10}$. Now,

$$\begin{aligned} |R_1 R_2| &= 90 \\ |R_1 R_3| &= 100 \\ |R_2 R_3| &= 100 \end{aligned}$$

and

$$\begin{aligned} C_{nl}(R_1 R_2 R_3) &= 10 * 10 + 90 * 100 = 9100 \\ C_{nl}(R_1 R_3 R_2) &= 10 * 100 + 100 * 10 = 2000 \\ C_{nl}(R_2 R_3 R_1) &= 10 * 100 + 100 * 10 = 2000 \end{aligned}$$

We see that $R_1 R_2 R_3$ has the smallest sum of intermediate result sizes but produces the highest cost. Hence, C_{nlj} is not ΣIR.

3.1.4 Classification of Join Ordering Problems

After having discussed the different classes of query graphs, join trees and cost functions, we can classify join ordering problems. To define a certain join ordering problem, we have to pick one entry from every class:

> Query Graph Classes $\times$ Possible Join Tree Classes $\times$ Cost Function Classes

The query graph classes considered are *chain*, *star*, *tree*, and *cyclic*. For the join tree classes we distinguish between the different join tree shapes, i.e. whether they are left-deep, zig-zag, or bushy trees. We left out the right-deep trees, since they do not differ in their behavior from left-deep trees. We also have to take into account whether cross products are considered or not. For cost functions, we use a simple classification: we only distinguish between those that have the ASI property and those that do not. This leaves us with $4 * 3 * 2 * 2 = 48$ different join ordering problems. For these, we will first review search space sizes and complexity. Then, we discuss several algorithms for join ordering. Last, we give some insight into cost distributions over the search space and how this might influence the benchmarking of different join ordering algorithms.

3.1.5 Search Space Sizes

Since search space sizes are easier to count if cross products are allowed, we consider them first. Then we turn to search spaces where cross products are not considered.

Join Trees with Cross Products We consider the number of join trees for a query graph with n relations. When cross products are allowed, the number of left-deep and right-deep join trees is $n!$. By allowing cross products, the query graph does not restrict the search space in any way. Hence, any of the $n!$ permutations of the n relations corresponds to a valid left-deep join tree. This is true independent of the query graph.

Similarly, the number of zig-zag trees can be estimated independently of the query graph. First note that for joining n relations, we need $n-1$ join operators. From any left-deep tree, we derive zig-zag trees by using the join's commutativity and exchange the left and right inputs. Hence, from any left-deep tree for n relations, we can derive 2^{n-2} zig-zag trees. We have to subtract another one, since the bottommost joins' arguments are exchanged in different left-deep trees. Thus, there exists a total of $2^{n-2}n!$ zig-zag trees. Again, this number is independent of the query graph.

The number of bushy trees can be estimated as follows. First, we need the number of binary trees. For n leaf nodes, the number of binary trees is given by $\mathcal{C}(n-1)$, where $\mathcal{C}(n)$ is defined by the recurrence

$$\mathcal{C}(n) = \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1)$$

with $\mathcal{C}(0) = 1$. The numbers $\mathcal{C}(n)$ are called the *Catalan Numbers* (see [206]). They can also be computed by the following formula:

$$\mathcal{C}(n) = \frac{1}{n+1}\binom{2n}{n}.$$

The Catalan Numbers grow in the order of $\Theta(4^n/n^{3/2})$.

After we know the number of binary trees with n leaves, we now have to attach the n relations to the leaves in all possible ways. For a given binary tree, this can be done in $n!$ ways. Hence, the total number of bushy trees is $n!\mathcal{C}(n-1)$. This can be simplified as follows (see also [291, 515, 835]):

$$\begin{aligned} n!\mathcal{C}(n-1) &= n!\frac{1}{n}\binom{2(n-1)}{n-1} \\ &= n!\frac{1}{n}\frac{(2n-2)!}{(n-1)!((2n-2)-(n-1))!} \\ &= \frac{(2n-2)!}{(n-1)!} \end{aligned}$$

Chain Queries, Left-Deep Join Trees, No Cartesian Product We now derive the function that calculates the number of left-deep join trees with no cross products for a chain query of n relations. That is, the query graph is $R_1 - R_2 - \ldots - R_{n-1} - R_n$. Let us denote the number of join trees by $f(n)$. Obviously, for $n = 0$ there is only one (the empty) join tree. For $n = 1$, there is also only one join tree (no join). For larger n: Consider the join trees for $R_1 - \ldots - R_{n-1}$ where relation R_{n-1} is the k-th relation from the bottom where k ranges from 1 to $n-1$. From such a join tree we can derive join trees for all n relations by adding relation R_n at any position following R_{n-1}.

There are $n-k$ such join trees. Only for $k=1$, we can also add R_n below R_{n-1}. Hence, for $k=1$ we have n join trees. How many join trees with R_{n-1} at position k are there? For $k=1$, R_{n-1} must be the first relation to be joined. Since we do not consider cross products, it must be joined with R_{n-2}. The next relation must be R_{n-3}, and so on. Hence, there is only one such join tree. For $k=2$, the first relation must be R_{n-2}, which is then joined with R_{n-1}. Then $R_{n-3},\ldots,R_1$ must follow in this order. Again, there is only one such join tree. For higher k, for R_{n-1} to occur safely at position k (no cross products) the $k-1$ relations $R_{n-2},\ldots,R_{n-k}$ must occur before R_{n-1}. There are exactly f($k-1$) join trees for the $k-1$ relations. On each such join tree we just have to add R_{n-1} on top of it to yield a join tree with R_{n-1} at position k.

Now we can compute the f(n) as $n+\sum_{k=2}^{n-1} f(k-1)*(n-k)$ for $n>1$. Solving this recurrence gives us $f(n)=2^{n-1}$. The proof is by induction. The case $n=1$ is trivial.

The induction step for $n>1$ provided by Thomas Neumann goes as follows:

$$\begin{aligned}
f(n) &= n+\sum_{k=2}^{n-1} f(k-1)*(n-k)\\
&= n+\sum_{k=0}^{n-3} f(k+1)*(n-k-2)\\
&= n+\sum_{k=0}^{n-3} 2^k*(n-k-2)\\
&= n+\sum_{k=1}^{n-2} k2^{n-k-2}\\
&= n+\sum_{k=1}^{n-2} 2^{n-k-2}+\sum_{k=2}^{n-2}(k-1)2^{n-k-2}\\
&= n+\sum_{i=1}^{n-2}\sum_{j=i}^{n-2} 2^{n-j-2}\\
&= n+\sum_{i=1}^{n-2}\sum_{j=0}^{n-i-2} 2^{j}\\
&= n+\sum_{i=1}^{n-2}(2^{n-i-1}-1)\\
&= n+\sum_{i=1}^{n-2} 2^{i}-(n-2)\\
&= n+(2^{n-1}-2)-(n-2)\\
&= 2^{n-1}
\end{aligned}$$

Chain Queries, Zig-Zag Join Trees, No Cartesian Product All possible zig-zag trees can be derived from a left-deep tree by exchanging the left and right arguments

of a subset of the joins. Since for the first join these alternatives are already considered within the set of left-deep trees, we are left with $n-2$ joins. Hence, the number of zig-zag trees for n relations in a chain query is $2^{n-2} * 2^{n-1} = 2^{2n-3}$.

Chain Queries, Bushy Join Trees, No Cartesian Product We can compute the number of bushy trees with no cross products for a chain query in the following way. Let us denote this number by f(n). Again, let us assume that the chain query has the form $R_1 - R_2 - \ldots - R_{n-1} - R_n$. For $n = 0$, we only have the empty join tree. For $n = 1$ we have one join tree. For $n = 2$ we have two join trees. For more relations, every subtree of the join tree must contain a subchain in order to avoid cross products. Further, the subchain can occur as the left or right argument of the join. Hence, we can compute f(n) as

$$\sum_{k=1}^{n-1} 2\,\mathrm{f}(k)\,\mathrm{f}(n-k)$$

This is equal to

$$2^{n-1}\,\mathcal{C}(n-1)$$

where $\mathcal{C}(n)$ are the Catalan Numbers. EX

Star Queries, No Cartesian Product The first join has to connect the center relation R_0 with any of the other relations. The other relations can follow in any order. Since R_0 can be the left or the right input of the first join, there are $2 * (n-1)!$ possible left-deep join trees for Star Queries with no Cartesian Product.

The number of zig-zag join trees is derived by exchanging the arguments of all but the first join in any left-deep join tree. We cannot consider the first join since we did so in counting left-deep join trees. Hence, the total number of zig-zag join trees is $2 * (n-1)! * 2^{n-2} = 2^{n-1} * (n-1)!$.

Constructing bushy join trees with no Cartesian Product from a Star Query other than zig-zag join trees is not possible.

Remarks The numbers for star queries are also upper bounds for tree queries. For clique queries, no join tree containing a cross product is possible. Hence, all join trees are valid join trees and the search space size is the same as the corresponding search space for join trees with cross products.

To give the reader a feeling for the numbers, the following tables contain the potential search space sizes for some n.

	Join trees without cross products				
	chain query			star query	
	left-deep	zig-zag	bushy	left-deep	zig-zag/bushy
n	2^{n-1}	2^{2n-3}	$2^{n-1}\mathcal{C}(n-1)$	$2*(n-1)!$	$2^{n-1}(n-1)!$
1	1	1	1	1	1
2	2	2	2	2	2
3	4	8	8	4	8
4	8	32	40	12	48
5	16	128	224	48	384
6	32	512	1344	240	3840
7	64	2048	8448	1440	46080
8	128	8192	54912	10080	645120
9	256	32768	366080	80640	10321920
10	512	131072	2489344	725760	185794560

	With cross products/clique		
	left-deep	zig-zag	bushy
n	$n!$	$2^{n-2}*n!$	$n!\mathcal{C}(n-1)$
1	1	1	1
2	2	2	2
3	6	12	12
4	24	96	120
5	120	960	1680
6	720	11520	30240
7	5040	161280	665280
8	40320	2580480	17297280
9	362880	46448640	518918400
10	3628800	928972800	17643225600

Note that in Figure 2.6 only 32 join trees are listed, whereas the number of bushy trees for chain queries with four relations is 40. The missing eight cases are those zig-zag trees which are symmetric (i.e. derived by applying commutativity to all occurring joins) to the ones contained in the second column.

From these numbers, it becomes immediately clear why historically the search space of query optimizers was restricted to left-deep trees and cross products for connected query graphs were not considered.

3.1.6 Problem Complexity

The complexity of the join ordering problem depends on several parameters. These are the shape of the query graph, the class of join trees to be considered, whether cross products are considered or not, and whether the cost function has the ASI property or not. Not for all the combinations complexity results are known. What is known is summarized in the following table.

Query graph	Join tree	Cross products	Cost function	Complexity
general	left-deep	no	ASI	NP-hard
tree/star/chain	left-deep	no	one join method (ASI)	P
star	left-deep	no	two join methods (NLJ+SMJ)	NP-hard
general/tree/star	left-deep	yes	ASI	NP-hard
chain	left-deep	yes	—	open
general	bushy	no	ASI	NP-hard
tree	bushy	no	—	open
star	bushy	no	ASI	P
chain	bushy	no	any	P
general	bushy	yes	ASI	NP-hard
tree/star/chain	bushy	yes	ASI	NP-hard

Ibaraki and Kameda were the first who showed that the problem of deriving optimal left-deep trees for cyclic queries is NP-hard for a cost function for an n-way nested loop join implementation [428]. The proof was repeated for the cost function C_{out} which has the ASI property [190, 842]. In both proofs, the clique problem was used for the reduction [311]. C_{out} was also used in the other proofs of NP-hardness results. The next line goes back to the same paper. Ibaraki and Kameda also described an algorithm to solve the join ordering problem for tree queries producing optimal left-deep trees for a special cost function for a nested-loop n-way join algorithm. Their algorithm was based on the observation that their cost function has the ASI property. For this case, they could derive an algorithm from an algorithm for a sequencing problem for job scheduling designed by Monma and Sidney [604]. They, in turn, used an earlier result by Lawler [520]. The algorithm of Ibaraki and Kameda was slightly generalized by Krishnamurthy, Boral, and Zaniolo, who were also able to sketch a more efficient algorithm. It improves the time bounds from $O(n^2 \log n)$ to $O(n^2)$. The disadvantage of both approaches is that with every relation, a fixed (i.e. join-tree independent) join implementation must be associated before the optimization starts. Hence, it only produces optimal trees if there is only one join implementation available or one is able to guess the optimal join method before hand. This might not be the case. The polynomial algorithm which we term IKKBZ is described in Section 3.2.2.

For star queries, Ganguly investigated the problem of generating optimal left-deep trees if no cross products but two different cost functions (one for nested loop join, the other for sort merge join) are allowed. It turned out that this problem is NP-hard [296].

The next line is due to Cluet and Moerkotte [190]. They showed by reduction from 3DM that taking into account cross products results in an NP-hard problem even for star queries. Remember that star queries are tree queries and general graphs.

The problem for general bushy trees follows from a result by Scheufele and Moerkotte [735]. They showed that building optimal bushy trees for cross products only (i.e. all selectivities equal one) is already NP-hard. This result also explains the last two lines.

By noting that for star queries, all bushy trees that do not contain a cross product are left-deep trees, the problem can be solved by the IKKBZ algorithm for left-deep trees. Ono and Lohman showed that for chain queries dynamic programming considers only a polynomial number of bushy trees if no cross products are considered [627].

This is discussed in Section 3.2.4.

The table is rather incomplete. Many open problems exist. For example, if we have chain queries and consider cross products: is the problem NP-hard or in P? Some results for this problem have been presented [735], but it is still an open problem (see Section 3.2.7). Open is also the case where we produce optimal bushy trees with no cross products for tree queries. Yet another example of an open problem is whether we could drop the ASI property and are still able to derive a polynomial algorithm for a tree query. This is especially important, since the cost function for a sort-merge algorithm does not have the ASI property.

Good summaries of complexity results for different join ordering problems can be found in the theses of Scheufele [733] and Hamalainen [388].

Given that join ordering is an inherently complex problem with no polynomial algorithm in sight, one might wonder whether there exists good polynomial approximation algorithms. Chances are that even this is not the case. Chatterji, Evani, Ganguly, and Yemmanuru showed that three different optimization problems — all asking for linear join trees — are not approximable [130].

3.2 Deterministic Algorithms

3.2.1 Heuristics

We now present some simple heuristic solutions to the problem of join ordering. These heuristics only produce left-deep trees. Since left-deep trees are equivalent with permutations, these heuristics order the joins according to some criterion.

The core algorithm for the heuristics discussed here is the *greedy algorithm* (for an introduction see [206]). In greedy algorithms, a *weight* is associated with each entity. In our case, weights are associated with each relation. A typical weight function is the cardinality of the relation ($|R|$). Given a weight function `weight`, a greedy join ordering algorithm works as follows:

```
GreedyJoinOrdering-1({R_1,...,R_n}, (*weight)(Relation))
Input:  a set of relations to be joined and a weight function
Output:  a join order
S = ε; // initialize S to the empty sequence
R = {R_1,...,R_n}; // let R be the set of all relations
while(!empty(R)) {
  Let k be such that:  weight(R_k) = min_{R_i∈R}(weight(R_i));
  R\ = R_k; // eliminate R_k from R
  S∘ = R_k; // append R_k to S
}
return S
```

This algorithm takes cross products into account. If we are only interested in left-deep join trees with no cross products, we have to require that R_k is connected to some of the relations contained in S in case $S \neq \epsilon$. Note that a more efficient implementation would sort the relations according to their weight.

Not all heuristics can be implemented with a greedy algorithm as simple as above. An often-used heuristics is to take the relation next that produces the smallest (next) intermediate result. This cannot be determined by the relation alone. One must take into account the sequence S already processed, since only then the selectivities of all predicates connecting relations in S and the new relation are deducible. And we must take the product of these selectivities and the cardinality of the new relation in order to get an estimate of the intermediate result's cardinality. As a consequence, the weights become *relative* to S. In other words, the weight function now has two parameters: a sequence of relations already joined and the relation whose relative weight is to be computed. Here is the next algorithm:

```
GreedyJoinOrdering-2({R_1, ..., R_n},
            (*weight)(Sequence of Relations, Relation))
Input:  a set of relations to be joined and a weight function
Output:  a join order
S = ε; // initialize S to the empty sequence
R = {R_1, ..., R_n}; // let R be the set of all relations
while(!empty(R)) {
  Let k be such that:  weight(S, R_k) = min_{R_i ∈ R}(weight(S, R_i));
  R\ = R_k; // eliminate R_k from R
  S○ = R_k; // append R_k to S
}
return S
```

Note that for this algorithm, sorting is not possible. `GreedyJoinOrdering-2` can be improved by taking every relation as the starting one.

```
GreedyJoinOrdering-3({R_1, ..., R_n}, (*weight)(Sequence of Relations, Relation))
Input:  a set of relations to be joined and a weight function
Output:  a join order
Solutions = ∅;
for (i = 1; i ≤ n; ++i) {
  S = R_i; // initialize S to a singleton sequence
  R = {R_1, ..., R_n} \ {R_i}; // let R be the set of all relations
  while(!empty(R)) {
    Let k be such that:  weight(S, R_k) = min_{R_i ∈ R}(weight(S, R_i));
    R\ = R_k; // eliminate R_k from R
    S○ = R_k; // append R_k to S
  }
  Solutions += S;
}
return cheapest in Solutions
```

In addition to the relative weight function mentioned before, another often used relative weight function is the product of the selectivities connecting relations in S with the new relation. This heuristics is sometimes called *MinSel*.

The above two algorithms generate linear join trees. Fegaras proposed a heuristic (named Greedy Operator Ordering (GOO)) to generate bushy join trees [265, 266]. The idea is as follows. A set of join trees `Trees` is initialized such that it contains all the relations to be joined. It then investigates all pairs of trees contained in `Trees`. Among all of these, the algorithm joins the two trees that result in the smallest intermediate result when joined. The two trees are then eliminated from `Trees` and the new join tree joining them is added to it. The algorithm then looks as follows:

```
GOO({R_1, ..., R_n})
Input:  a set of relations to be joined
Output:  join tree
Trees := {R_1, ..., R_n}
while (|Trees| != 1) {
  find T_i, T_j ∈ Trees such that i ≠ j, |T_i ⋈ T_j| is minimal
       among all pairs of trees in Trees
  Trees −= T_i;
  Trees −= T_j;
  Trees += T_i ⋈ T_j;
}
return the tree contained in Trees;
```

Our GOO variant differs slightly from the one proposed by Fegaras. He uses arrays, explicitly handles the forming of the join predicates, and materializes intermediate result sizes. Hence, his algorithm is a little more elaborated, but we assume that the reader is able to fill in the gaps.

None of our algorithms so far considers different join implementations. An explicit consideration of commutativity for non-symmetric cost functions could also help to produce better join trees. The reader is asked to work out the details of these extensions. In general, the heuristics do not produce the optimal plan. The reader is advised to find examples where the heuristics are far off the best possible plan.

EX
EX

3.2.2 Determining the Optimal Join Order in Polynomial Time

Since the general problem of join ordering is NP-hard, we cannot expect to find a polynomial solution for it. However, for special cases, we can expect to find solutions that work in polynomial time. These solutions can also be used as heuristics for the general case, either to find a not-that-bad join tree or to determine an upper bound for the costs that is then fed into a search procedure to prune the search space.

The most general case for which a polynomial solution is known is charactized by the following features:

- the query graph must be acyclic
- no cross products are considered

- the search space is restricted to left-deep trees
- the cost function must have the ASI property

The algorithm was presented by Ibaraki and Kameda [428]. Later Krishnamurthy, Boral, and Zaniolo presented it again for some other cost functions (still having the ASI property) [503]. They also observed that the upper bound $O(n^2 \log n)$ of the original algorithm could be improved to $O(n^2)$. In any case, the algorithm is based on an algorithm discovered by Monma and Sidney for job scheduling [520, 604] . Let us call the (unimproved) algorithm IKKBZ-Algorithm.

The IKKBZ-Algorithm considers only join operations that have a cost function of the form:

$$cost(R_i \bowtie R_j) = |R_i| * h_j(|R_j|)$$

where each R_j can have its own cost function h_j. We denote the set of h_j by H and parameterize cost functions with it. Example instanciations are

- $h_j \equiv 1.2$ for main memory hash-based joins
- $h_j \equiv$ id for nested-loop joins

where id is the identity function. Let us denote by n_i the cardinality of the relation R_i ($n_i := |R_i|$). Then, the $h_i(n_i)$ represent the costs per input tuple to be joined with R_i.

The algorithm works as follows. For every relation R_k it computes the optimal join order under the assumption that R_k is the first relation in the join sequence. The resulting subproblems then resemble a job-scheduling problem that can be solved by the Monma-Sidney-Algorithm [604].

In order to present this algorithm, we need the notion of a *precedence graph*. A *precedence graph* is formed by taking a node in the (undirected) query graph and making this node a root node of a (directed) tree where the edges point away from the selected root node. Hence, for acyclic, connected query graphs—those we consider in this section—a precedence graph is a tree. We construct the precedence graph of a query graph $G = (V, E)$ as follows:

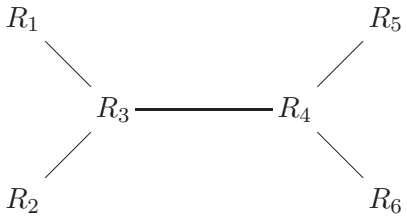
- Make some relation $R_k \in V$ the root node of the precedence graph.
- As long as not all relations are included in the precedence graph: Choose a relation $R_i \in V$, such that $(R_j, R_i) \in E$ is an edge in the query graph and R_j is already contained in the (partial) precedence graph constructed so far and R_i is not. Add R_j and the edge $R_j \rightarrow R_i$ to the precedence graph.

A sequence $S = v_1, \ldots, v_k$ of nodes conforms to a precedence graph $G = (V, E)$ if the following conditions are satisfied:

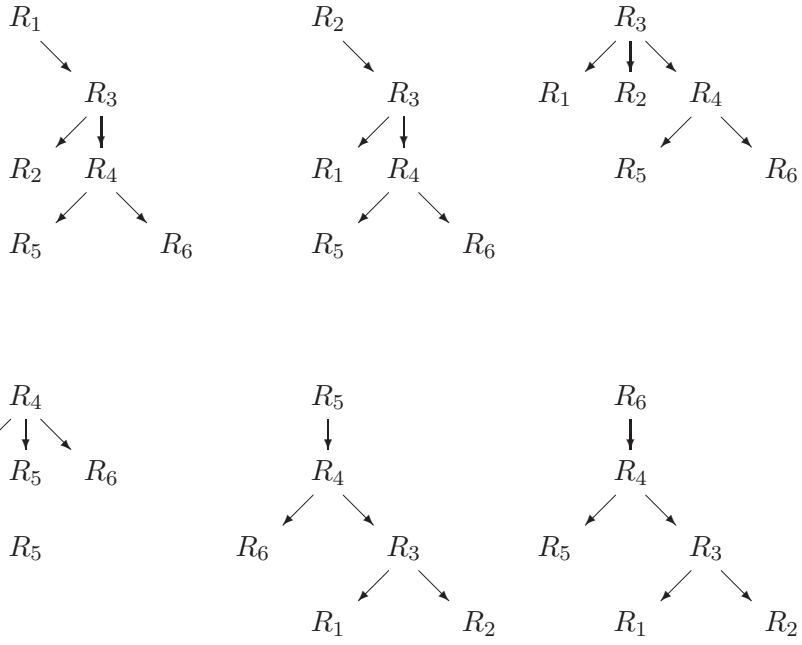
1. for all i ($2 \leq i \leq k$) there exists a j ($1 \leq j < i$) with $(v_j, v_i) \in E$ and
2. there is no edge $(v_i, v_j) \in E$ for $i > j$.

For non-empty sequences U and V in a precedence graph, we write $U \rightarrow V$ if, according to the precedence graph, U must occur before V. This requires U and V to be disjoint. More precisely, there can only be paths from nodes in U to nodes in V and at least one such path exists.

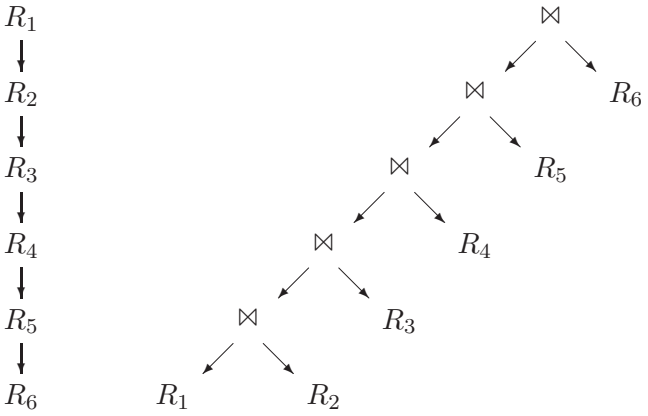
Consider the following query graph:



For this query graph, we can derive the following precedence graphs:



The IKKBZ-Algorithm takes a single precedence graph and produces a new one that is totally ordered. From this total order it is very easy to construct a corresponding join graph: the following figure contains a totally ordered precedence graph (left-hand side) as generated by the IKKBZ-Algorithm and the corresponding join graph on the right-hand side.



Define

$$\begin{aligned} R_{1,2,\ldots,k} &:= R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k \\ n_{1,2,\ldots,k} &:= |R_{1,2,\ldots,k}| \end{aligned}$$

For a given precedence graph, let R_i be a relation and $\mathcal{R}_i$ be the set of relations from which there exists a path to R_i. Then, in any join tree adhering to the precedence graph, all relations in $\mathcal{R}_i$ and only those will be joined before R_i. Hence, we can define $s_i = \prod_{R_j \in \mathcal{R}_i} f_{i,j}$ for $i > 1$. Note that for any i only one j with $f_{i,j} \neq 1$ exists in the product. If the precedence graph is a chain, then the following holds:

$$n_{1,2,\ldots,k+1} = n_{1,2\ldots,k} * s_{k+1} * n_{k+1}$$

We define $s_1 = 1$. Then we have

$$n_{1,2} = s_2 * (n_1 * n_2) = (s_1 * s_2) * (n_1 * n_2)$$

and, in general,

$$n_{1,2,\ldots,k} = \prod_{i=1}^{k} (s_i * n_i).$$

We call the s_i selectivities, although they depend on the precedence graph.

The costs for a totally ordered precedence graph G can thus be computed as follows:

$$\begin{aligned} Cost_H(G) &= \sum_{i=2}^{n} [n_{1,2,\ldots,i-1} * h_i(n_i)] \\ &= \sum_{i=2}^{n} [(\prod_{j=1}^{i-1} s_j * n_j) * h_i(n_i)] \end{aligned}$$

If we define $h_i(n_i) = s_i n_i$, then C$ost_H \equiv C_{\text{out}}$. The factor $s_i n_i$ determines by how much the input relation to be joined with R_i changes its cardinality after the join has been performed. If $s_i n_i$ is less than one, we call the join *decreasing*, if it is larger than one, we call the join *increasing*. This distinction plays an important role in the heuristic discussed in Section 3.2.3.

The cost function can also be defined recursively.

Definition 3.2.1 *Define the cost function C_H as follows:*

$$\begin{aligned} C_H(\epsilon) &= 0 \\ C_H(R_j) &= 0 \quad \textit{if } R_j \textit{ is the root} \\ C_H(R_j) &= h_j(n_j) \quad \textit{else} \\ C_H(S_1 S_2) &= C_H(S_1) + T(S_1) * C_H(S_2) \end{aligned}$$

where

$$\begin{aligned} T(\epsilon) &= 1 \\ T(S) &= \prod_{R_i \in S} (s_i * n_i) \end{aligned}$$

It is easy to prove by induction that C_H is well-defined and that $C_H(G) = Cost_H(G)$. EX

Definition 3.2.2 *Let A and B be two sequences and V and U two non-empty sequences. We say that a cost function C has the* adjacent sequence interchange property *(ASI property) if and only if there exists a function T and a rank function defined for sequences S as*

$$rank(S) = \frac{T(S) - 1}{C(S)}$$

such that for non-empty sequences $S = AUVB$ the following holds

$$C(AUVB) \leq C(AVUB) \quad \prec\!\succ \quad rank(U) \leq rank(V) \tag{3.1}$$

if $AUVB$ and $AVUB$ satisfy the precedence constraints imposed by a given precedence graph.

Lemma 3.2.3 *The cost function C_H defined in Definition 3.2.1 has the ASI property.*

The proof is very simple. Using the definition of C_H, we have

$$\begin{aligned} C_H(AUVB) &= C_H(A) \\ &\quad + T(A)C_H(U) \\ &\quad + T(A)T(U)C_H(V) \\ &\quad + T(A)T(U)T(V)C_H(B) \end{aligned}$$

and, hence,

$$\begin{aligned} C_H(AUVB) - C_H(AVUB) &= T(A)[C_H(V)(T(U) - 1) - C_H(U)(T(V) - 1)] \\ &= T(A)C_H(U)C_H(V)[rank(U) - rank(V)] \end{aligned}$$

The proposition follows. □

Definition 3.2.4 *Let $M = \{A_1, \ldots, A_n\}$ be a set of node sequences in a given precedence graph. Then, M is a called a* module *if for all sequences B that do not overlap with the sequences in M one of the following conditions holds:*

- $B \rightarrow A_i, \forall\, 1 \leq i \leq n$
- $A_i \rightarrow B, \forall\, 1 \leq i \leq n$
- $B \not\rightarrow A_i$ *and* $A_i \not\rightarrow B, \forall\, 1 \leq i \leq n$

Lemma 3.2.5 *Let C be any cost function with the ASI property and $\{A, B\}$ a module. If $A \rightarrow B$ and additionally $rank(B) \leq rank(A)$, then we find an optimal sequence among those in which B directly follows A.*

Proof Every optimal permutation must have the form (U, A, V, B, W), since $A \rightarrow B$. Assumption: $V \neq \epsilon$. If $rank(V) \leq rank(A)$, then we can exchange V and A without increasing the costs. If $rank(A) \leq rank(V)$, we have $rank(B) \leq rank(V)$ due to the transitivity of $\leq$. Hence, we can exchange B and V without increasing the costs. Both exchanges produce legal sequences obeying the precedence graph, since $\{A, B\}$ is a module. □

If the precedence graph demands $A \rightarrow B$ but $rank(B) \leq rank(A)$, we speak of *contradictory sequences* A and B. Since the lemma shows that no non-empty subsequence can occur between A and B, we will combine A and B into a new single node replacing A and B. This node represents a *compound relation* comprising all relations in A and B. Its cardinality is computed by multiplying the cardinalities of all relations occurring in A and B, and its selectivity s is the product of all the selectivities s_i of the relations R_i contained in A and B. The continued process of this step until no more contradictory sequence exists is called *normalization*. The opposite step, replacing a compound node by the sequence of relations it was derived from, is called *denormalization*.

We can now present the algorithm `IKKBZ`.

```
IKKBZ(G)
Input: an acyclic query graph G for relations R_1,...,R_n
Output: the best left-deep tree
R = ∅;
for (i = 1; i ≤ n; ++i) {
  Let G_i be the precedence graph derived from G and rooted at R_i;
  T = IKKBZ-Sub(G_i);
  R+ = T;
}
return best of R;

IKKBZ-Sub(G_i)
Input: a precedence graph G_i for relations R_1,...,R_n rooted at some R_i
Output: the optimal left-deep tree under G_i
while (G_i is not a chain) {
  let r be the root of a subtree in G_i whose subtrees are chains;
  IKKBZ-Normalize(r);
  merge the chains under r according to the rank function
    in ascending order;
}
IKKBZ-Denormalize(G_i);
return G_i;

IKKBZ-Normalize(r)
Input: the root r of a subtree T of a precedence graph G = (V, E)
Output: a normalized subchain
while (∃ r', c ∈ V, r →* r', (r', c) ∈ E: rank(r') > rank(c)) {
  replace r' by a compound relation r'' that represents r'c;
```

```
};
```

We do not give the details of `IKKBZ-Denormalize`, as it is trivial.

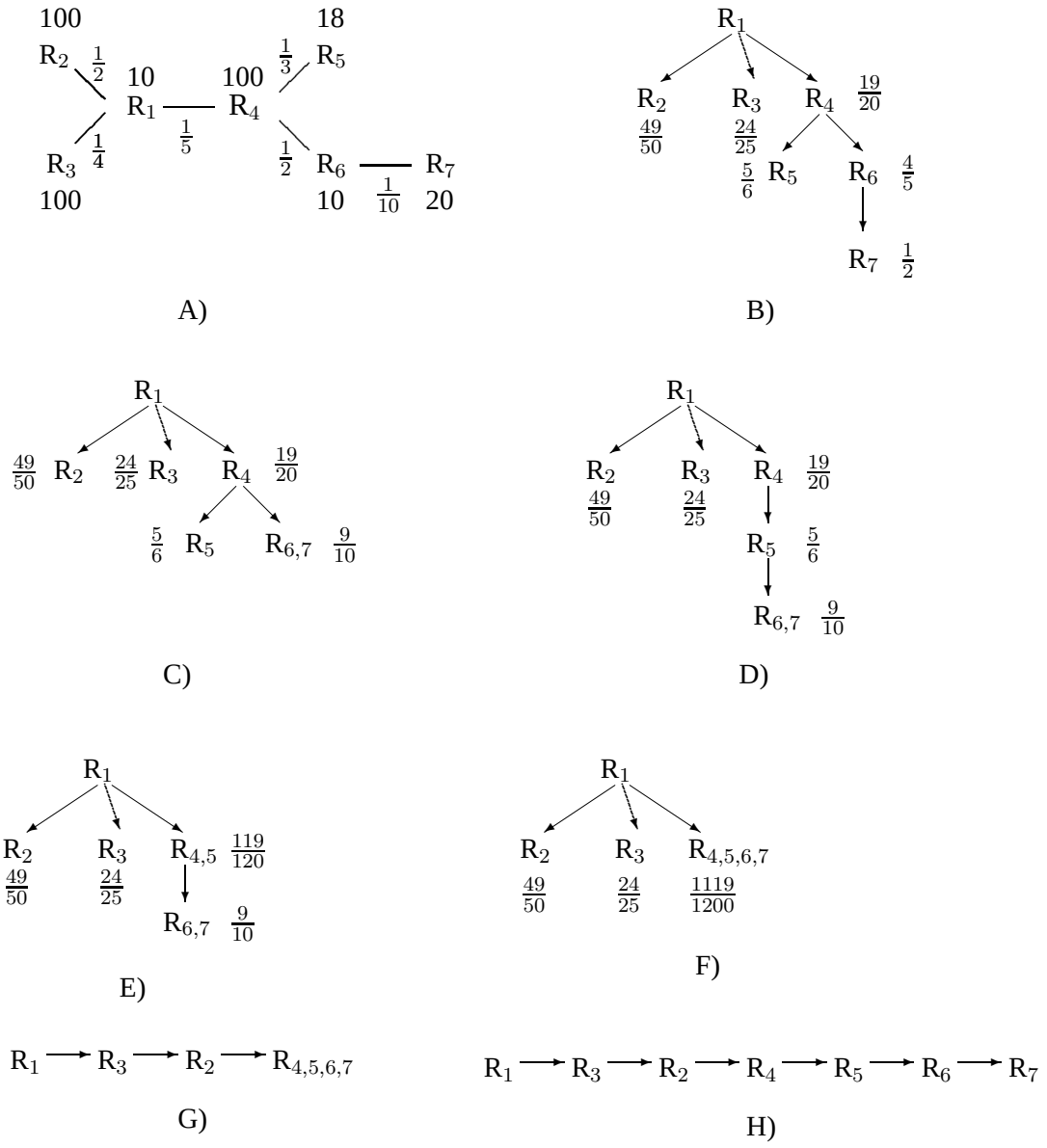


Figure 3.3: Illustrations for the IKKBZ Algorithm

Let us illustrate the algorithm `IKKBZ-Sub` by a simple example. We use the cost function C_{out}. Figure 3.3 A) shows a query graph. The relations are annotated with their sizes and the edges with the join selectivities. Chosing R_1 as the root of the precedence graph results in B). There, the nodes are annotated by the ranks of the relations. R_4 is the root of a subtree all of whose subtrees are chains. Hence, we normalize it. For R_5, there is nothing to do. The ranks of R_6 and R_7 are contradictory. We form a compound relation $R_{6,7}$, calculate its cardinality, selectivity, and rank. The

latter is shown in C). Merging the two subchains under R_4 results in D). Now R_1 is the root of a subtree with only chains underneath. Normalization detects that the ranks for R_4 and R_5 are contradictory. E) shows the tree after introducing the compound relation $R_{4,5}$. Now $R_{4,5}$ and $R_{6,7}$ have contradictory ranks, and we replace them by the compound relation $R_{4,5,6,7}$ as shown in F). Merging the chains under R_1 gives G). Since this is a chain, we leave the loop and denormalize. The final result is shown in H).

We can use the IKKBZ-Algorithm to derive a heuristics also for cyclic queries, i.e. for general query graphs. In a first step, we determine a minimal spanning tree of the query graph. It is then used as the input query graph for the IKKBZ-Algorithm. Let us call this the *IKKBZ-based Heuristics*.

3.2.3 The Maximum-Value-Precedence Algorithm

Lee, Shih, and Chen proposed a very interesting heuristics for the join ordering problem [521]. They use a *weighted directed join graph* (WDJG) to represent queries. Within this graph, every join tree corresponds to a spanning tree. Given a conjunctive query with join predicates P. For a join predicate $p \in P$, we denote by $\mathcal{R}(p)$ the relations whose attributes are mentioned in p.

Definition 3.2.6 *The* directed join graph *of a conjunctive query with join predicates P is a triple $G = (V, E_p, E_v)$, where V is the set of nodes and E_p and E_v are sets of directed edges defined as follows. For any two nodes $u, v \in V$, if $\mathcal{R}(u) \cap \mathcal{R}(v) \neq \emptyset$ then $(u, v) \in E_p$ and $(v, u) \in E_p$. If $\mathcal{R}(u) \cap \mathcal{R}(v) = \emptyset$, then $(u, v) \in E_v$ and $(v, u) \in E_v$. The edges in E_p are called* physical edges*, those in E_v* virtual edges*.*

Note that in G for every two nodes u, v, there is an edge (u, v) that is either physical or virtual. Hence, G is a clique.

Let us see how we can derive a join tree from a spanning tree of a directed join graph. Figure 3.4 I) gives a simple query graph Q corresponding to a chain and Part II) presents Q's directed join graph. Physical edges are drawn by solid arrows, virtual edges by dotted arrows. Let us first consider the spanning tree shown in Part III a). It says that we first execute $R_1 \bowtie_{p_{1,2}} R_2$. The next join predicate to evaluate is $p_{2,3}$. Obviously, it does not make much sense to execute $R_2 \bowtie_{p_{2,3}} R_3$, since R_1 and R_2 have already been joined. Hence, we replace R_2 in the second join by the result of the first join. This results in the join tree $(R_1 \bowtie_{p_{1,2}} R_2) \bowtie_{p_{2,3}} R_3$. For the same reason, we proceed by joining this result with R_4. The final join tree is shown in Part III b). Part IV a) shows another spanning tree. The two joins $R_1 \bowtie_{p_{1,2}} R_2$ and $R_3 \bowtie_{p_{3,4}} R_4$ can be executed independently and do not influence each other. Next, we have to consider $p_{2,3}$. Both R_2 and R_3 have already been joined. Hence, the last join processes both intermediate results. The final join tree is shown in Part IV b). The spanning tree shown in Part V a) results in the same join tree shown in Part V b). Hence, two different spanning trees can result in the same join tree. However, the spanning tree in Part IV a) is more specific in that it demands $R_1 \bowtie_{p_{1,2}} R_2$ to be executed before $R_3 \bowtie_{p_{3,4}}$.

Next, take a look at Figure 3.5. Part I), II), and III a) show a query graph, its directed join tree and a spanning tree. To build a join tree from the spanning tree we proceed as follows. We have to execute $R_2 \bowtie_{p_{2,3}} R_3$ and $R_3 \bowtie R_4$ first. In which way

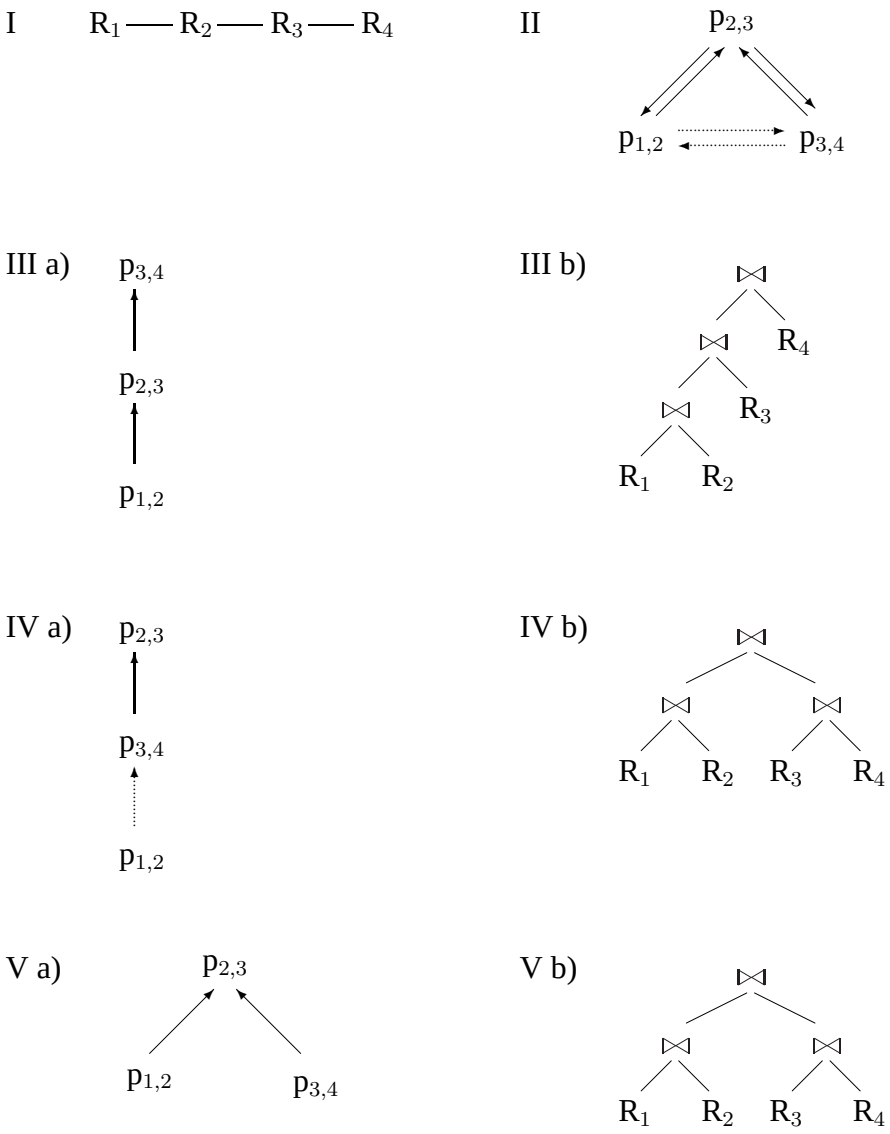


Figure 3.4: A query graph, its directed join graph, some spanning trees and join trees

we do so is not really fixed by the spanning tree. So let us do both in parallel. Next is $p_{1,2}$. The only dependency the spanning tree gives us is that it should be executed after $p_{3,4}$. Since there is no common relation between those two, we perform $R_1 \bowtie_{p_{1,2}} R_2$. Last is $p_{4,5}$. Since we find $p_{3,4}$ below it, we use the intermediate result produced by it as a replacement for R_4. The result is shown in Part III b). It has three loose ends. Additional joins are required to tie the partial results together. Obviously, this is not what we want. A spanning tree that avoids this problem of additional joins is called *effective*. It can be shown that a spanning tree $T = (V, E)$ is effective if it satisfies the following conditions [521]:

1. T is a binary tree,

2. for all inner nodes v and node u with $(u, v) \in E$ it holds that $\mathcal{R}^*(T(u)) \cap$

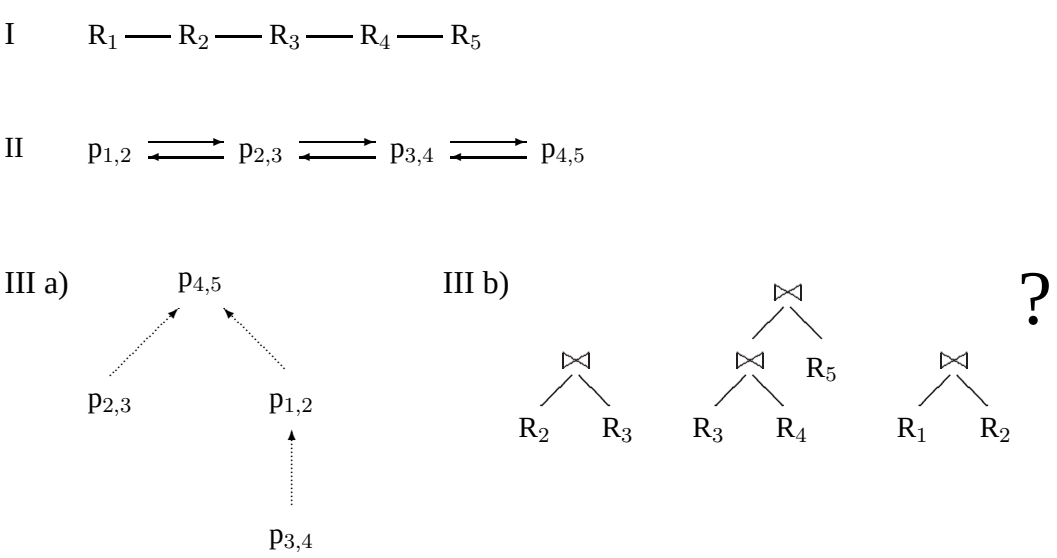


Figure 3.5: A query graph, its directed join tree, a spanning tree and its problem

$\mathcal{R}(v) \neq \emptyset$, and

3. for all nodes v, u_1, u_2 with $u_1 \neq u_2$, $(u_1, v) \in E$, and $(u_2, v) \in E$ one of the following two conditions holds:

   (a) $((\mathcal{R}^*(T(u_1)) \cap \mathcal{R}(v)) \cap (\mathcal{R}^*(T(u_2)) \cap \mathcal{R}(v))) = \emptyset$ or

   (b) $(\mathcal{R}^*(T(u_1)) \cap \mathcal{R}(v) = \mathcal{R}(v)) \vee (\mathcal{R}^*(T(u_2)) \cap \mathcal{R}(v) = \mathcal{R}(v))$.

Thereby, we denote by $T(v)$ the partial tree rooted at v and by $\mathcal{R}^*(T') = \cup_{v \in T'} \mathcal{R}(v)$ the set of all relations in subtree T'.

We see that the spanning tree in Figure 3.5 III a) is ineffective since, for example, $\mathcal{R}(p_{2,3}) \cap \mathcal{R}(p_{4,5}) = \emptyset$. The spanning tree in Figure 3.4 IV a) is also ineffective. During the algorithm we will take care—by checking the above conditions—that only effective spanning trees are generated.

We now assign weights to the edges of the directed join graph. For two nodes $v, u \in V$ define $u \sqcap v := \mathcal{R}(u) \cap \mathcal{R}(v)$. For simplicity, we assume that every predicate involves exactly two relations. Then for all $u, v \in V$, $u \sqcap v$ contains a single relation. Let $v \in V$ be a node with $\mathcal{R}(v) = \{R_i, R_j\}$. We abbreviate $R_i \bowtie_v R_j$ by $\bowtie_v$. Using these notations, we can attach weights to the edges to define the *weighted directed join graph*.

Definition 3.2.7 *Let $G = (V, E_p, E_v)$ be a directed join graph for a conjunctive query with join predicates P. The* weighted directed join graph *is derived from G by attaching a weight to each edge as follows:*

- *Let $(u, v) \in E_p$ be a physical edge. The weight $w_{u,v}$ of (u, v) is defined as*

$$w_{u,v} = \frac{|\bowtie_u|}{|u \sqcap v|}.$$

- *For virtual edges $(u, v) \in E_v$, we define $w_{u,v} = 1$.*

(Lee, Shih, and Chen actually attach two weights to each edge: one additional weight for the size of the tuples (in bytes) [521].)

The weights of physical edges are equal to the s_i of the dependency graph used in the IKKBZ-Algorithm (Section 3.2.2). To see this, assume $\mathcal{R}(u) = \{R_1, R_2\}$, $\mathcal{R}(v) = \{R_2, R_3\}$. Then

$$\begin{aligned} w_{u,v} &= \frac{|\bowtie_u|}{|u \sqcap v|} \\ &= \frac{|R_1 \bowtie_u R_2|}{|R_2|} \\ &= \frac{f_{1,2}\,|R_1|\,|R_2|}{|R_2|} \\ &= f_{1,2}\,|R_1| \end{aligned}$$

Hence, if the join $R_1 \bowtie_u R_2$ is executed before the join $R_2 \bowtie_v R_3$, the input size to the latter join changes by a factor $w_{u,v}$. This way, the influence of a join on another join is captured by the weights. Since those nodes connected by a virtual edge do not influence each other, a weight of 1 is appropriate.

Additionally, we assign weights to the nodes of the directed join graph. The weight of a node reflects the change in cardinality to be expected when certain other joins have been executed before. They are specified by a (partial) spanning tree S. Given S, we denote by $\bowtie^S_{p_{i,j}}$ the result of the join $\bowtie_{p_{i,j}}$ if all joins preceding $p_{i,j}$ in S have been executed. Then the weight attached to node $p_{i,j}$ is defined as

$$w(p_{i,j}, S) = \frac{|\bowtie^S_{p_{i,j}}|}{|R_i \bowtie_{p_{i,j}} R_j|}.$$

For empty sequences ϵ, we define $w(p_{i,j}, \epsilon) = |R_i \bowtie_{p_{i,j}} R_j|$. Similarly, we define the cost of a node $p_{i,j}$ depending on other joins preceding it in some given spanning tree S. We denote this by `cost(`$p_{i,j}, S$`)`. The actual cost function can be one we have introduced so far or any other one. In fact, if we have a choice of several join implementations, we can take the minimum over all their cost functions. This then choses the most effective join implementation.

The maximum value precedence algorithm works in two phases. In the first phase, it searches for edges with a weight smaller than one. Among these, the one with the biggest impact is chosen. This one is then added to the spanning tree. In other words, in this phase, the costs of expensive joins are minimized by making sure that *(size) decreasing joins* are executed first. The second phase adds edges such that the intermediate result sizes increase as little as possible.

`MVP(`G`)`
Input: `a weighted directed join graph` $G = (V, E_p, E_v)$
Output: `an effective spanning tree`
Q_1`.insert(`V`); /* priority queue with smallest node weights` $w(\cdot)$ `first */`
$Q_2 = \emptyset$`; /* priority queue with largest node weights` $w(\cdot)$ `first */`
$G' = (V', E')$ `with` $V' = V$ `and` $E' = E_p$`; /* working graph */`

```
S = (V_S, E_S) with V_S = V and E_S = ∅; /* resulting effective spanning tree */
while (!Q_1.empty() && |E_S| < |V| − 1) { /* Phase I */
  v = Q_1.head();
  among all (u, v) ∈ E', w_{u,v} < 1 such that
    S' = (V, E'_S) with E'_S = E_S ∪ {(u, v)} is acyclic and effective
    select one that maximizes cost(⋈_v, S) - cost(⋈_v, S');
  if (no such edge exists) {
    Q_1.remove(v);
    Q_2.insert(v);
    continue;
  }
  MvpUpdate((u, v));
  recompute w(·) for v and its ancestors; /* rearranges Q_1 */
}
while (!Q_2.empty() && |E_S| < |V| − 1) { /* Phase II */
  v = Q_2.head();
  among all (u, v), (v, u) ∈ E' denoted by (x, y) henceforth
    such that
    S' = (V, E'_S) with E'_S = E_S ∪ {(x, y)} is acyclic and effective
    select the one that minimizes cost(⋈_v, S') - cost(⋈_v, S);
  MvpUpdate((x, y));
  recompute w(·) for y and its ancestors; /* rearranges Q_2 */
}
return S;

MvpUpdate((u, v))
Input: an edge to be added to S
Output: side-effects on S, G',
  E_S ∪= {(u, v)};
  E' \= {(u, v), (v, u)};
  E' \= {(u, w)|(u, w) ∈ E'}; /* (1) */
  E' ∪= {(v, w)|(u, w) ∈ E_p, (v, w) ∈ E_v}; /* (3) */
  if (v has two inflowing edges in S) { /* (2) */
    E' \= {(w, v)|(w, v) ∈ E'};
  }
  if (v has one outflowing edge in S) { /* (1) in paper but not needed */
    E' \= {(v, w)|(v, w) ∈ E'};
  }
```

Note that in order to test for the effectiveness of a spanning tree in the algorithm, we just have to check the conditions for the node the selected edge leads to.

`MvpUpdate` first adds the selected edge to the spanning tree. It then eliminates edges that need not to be considered for building an effective spanning tree. Since (u, v) has been added, both (u, v) and (v, u) do not have to be considered any longer. Also, since effective spanning trees are binary trees, (1) every node must have only one parent node and (2) at most two child nodes. The edges leading to a violation are

eliminated by `MvpUpdate` in the lines commented with the corresponding numbers. For the line commented (3) we have the situation that $u \rightarrow v \dashrightarrow w$ and $u \rightarrow w$ in G. This means that u and w have common relations, but v and w do not. Hence, the result of performing v on the result of u will have a common relation with w. Thus, we add a (physical) edge $v \rightarrow w$.

3.2.4 Dynamic Programming

Algorithms

Consider the two join trees

$$(((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) \bowtie R_5$$

and

$$(((R_3 \bowtie R_1) \bowtie R_2) \bowtie R_4) \bowtie R_5.$$

If we know that $((R_1 \bowtie R_2) \bowtie R_3)$ is cheaper than $((R_3 \bowtie R_1) \bowtie R_2)$, we know that the first join tree is cheaper than the second. Hence, we could avoid generating the second alternative and still won't miss the optimal join tree. The general principle behind this is the *optimality principle* (see [205]). For the join ordering problem, it can be stated as follows.[1]

> Let T be an optimal join tree for relations $R_1, \ldots, R_n$. Then, every subtree S of T must be an optimal join tree for the relations it contains.

To see why this holds, assume that the optimal join tree T for relations $R_1, \ldots, R_n$ contains a subtree S which is not optimal. That is, there exists another join tree S' for the relations contained in S with strictly lower costs. Denote by T' the join tree derived by replacing S in T by S'. Since S' contains the same relations as S, T' is a join tree for the relations $R_1, \ldots, R_n$. The costs of the join operators in T and T' that are not contained in S and S' are the same. Then, since the total cost of a join tree is the sum of the costs of the join operators and S' has lower costs than S, T' has lower costs than T. This contradicts the optimality of T.

The idea of dynamic programming applied to the generation of optimal join trees now is to generate optimal join trees for subsets of $R_1, \ldots, R_n$ in a bottom-up fashion. First, optimal join trees for subsets of size one, i.e. single relations, are generated. From these, optimal join trees of size two, three and so on until n are generated.

Let us first consider generating optimal left-deep trees. There, join trees for subsets of size k are generated from subsets of size $k-1$ by adding a new join operator whose left argument is a join tree for $k-1$ relations and whose right argument is a single relation. Exchanging left and right gives us the procedure for generating right-deep trees. If we want to generate zig-zag trees since our cost function is asymmetric, we have to consider both alternatives and take the cheapest one. We capture this in a procedure `CreateJoinTree` that takes two join trees as arguments and generates the above-mentioned alternatives. In case we want to consider different implementations for the join, we have to perform the above steps for all of them and return the cheapest alternative. Summarizing, the pseudo-code for `CreateJoinTree` looks as follows:

[1] The optimality principle does not hold in the presence of properties.

```
CreateJoinTree(T1, T2)
Input:  two (optimal) join trees T1 and T2.
          for linear trees, we assume that T2 is a single relation
Output:  an (optimal) join tree for joining T1 and T2.
BestTree = NULL;
for all implementations impl do {
  if(!RightDeepOnly) {
    Tree = T1 ⋈^impl T2
    if (BestTree == NULL || cost(BestTree) > cost(Tree)) {
      BestTree = Tree;
    }
  }
  if(!LeftDeepOnly) {
    Tree = T2 ⋈^impl T1
    if (BestTree == NULL || cost(BestTree) > cost(Tree)) {
      BestTree = Tree;
    }
  }
}
return BestTree;
```

The boolean variables `RightDeepOnly` and `LeftDeepOnly` are used to restrict the search space to right-deep trees and left-deep trees. If both are false, zig-zag trees are generated. However, `CreateJoinTree` also generates bushy trees, if none of the input trees is a single relation.

In case of linear trees, T_2 will be the single relation in all of our algorithms. `CreateJoinTree` should not copy T_1 or T_2. Instead, the newly generated join trees should share T_1 and T_2 by using pointers. Further, the join trees generated do not really need to be generated except for the final (best) join tree: the cost functions should be implemented such that they can be evaluated if they are given the left and right argument of the join.

Using `CreateJoinTree`, we are now ready to present our first dynamic programming algorithm in pseudo-code.

```
DP-Linear-1({R1, ..., Rn})
Input:  a set of relations to be joined
Output:  an optimal left-deep (right-deep, zig-zag) join tree
for (i = 1; i <= n; ++i) {
  BestTree({Ri}) = Ri;
}
for (i = 1; i < n; ++i) {
  for all S ⊆ {R1, ..., Rn}, |S| = i do {
    for all Rj ∈ {R1, ..., Rn}, Rj ∉ S do {
```

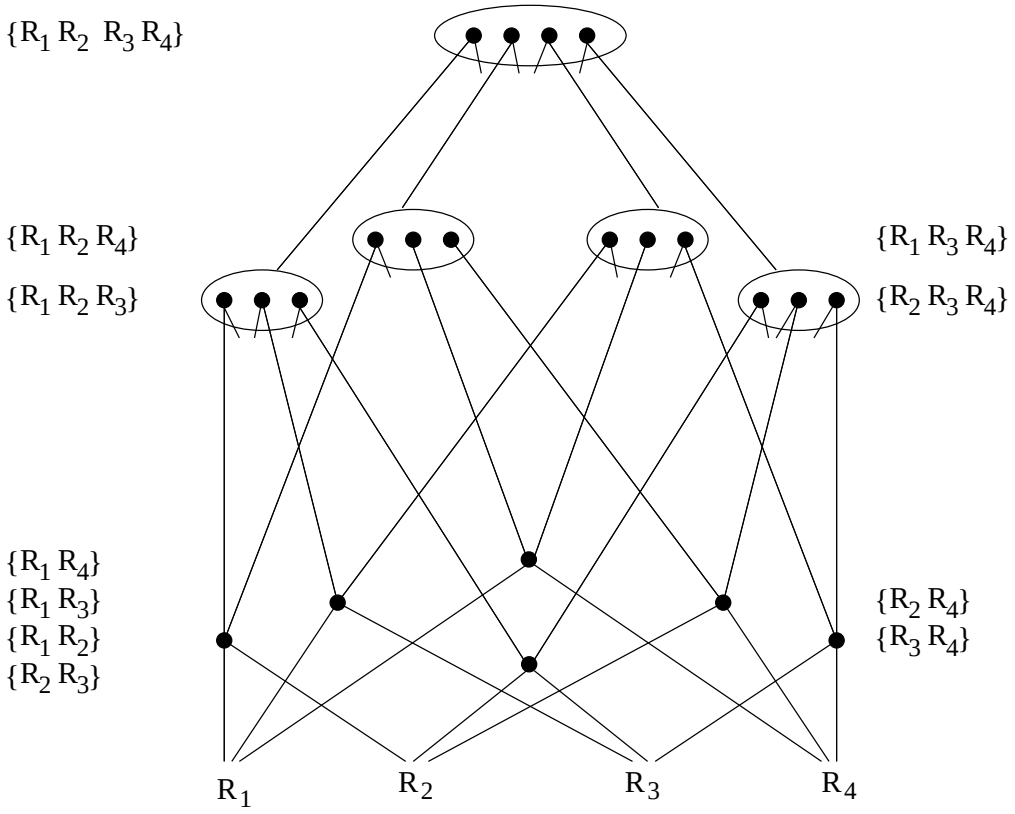


Figure 3.6: Search space with sharing under optimality principle

```
        if (NoCrossProducts && !connected({R_j}, S)) {
            continue;
        }
        CurrTree = CreateJoinTree(BestTree(S),R_j);
        S' = S ∪ {R_j};
        if (BestTree(S') == NULL || cost(BestTree(S')) > cost(CurrTree)) {
            BestTree(S') = CurrTree;
        }
    }
  }
}
return BestTree({R_1, ..., R_n});
```

`NoCrossProducts` is a boolean variable indicating whether cross products should be investigated. Of course, if the join graph is not connected, there must be a cross product, but for `DP-Linear-1` and subsequent algorithms we assume that it is connected. The boolean function `connected` returns true, if there is a join predicate between one of the relations in its first argument and one of the relations in its second. The variable `BestTree` keeps track of the best join trees generated for every subset of the relations $\{R_1, \ldots, R_n\}$. How this is done may depend on several parameters. The approaches are to use a hash table or an array of size $2^n(-1)$. Another issue is how to represent the sets of relations. Typically, bitvector representations are used.

XC search space size difference problem

Then, testing for membership, computing a set's complement, adding elements and unioning is cheap. Yet another issue is the order in which join trees are generated. The procedure `DP-Linear-1` takes the approach to generate the join trees for subsets of size $1, 2, \ldots, n$. To do so, it must be able to access the subsets of $\{R_1, \ldots, R_n\}$ or their respective join trees by their size. One possibility is to chain all the join trees for subsets of a given size k ($1 \leq k \leq n$) and to use an array of size n to keep pointers to the start of the lists. In this case, to every join tree the set of relations it contains is attached, in order to be able to perform the test $R_i \notin S$. One way to do this is to embed a bitvector into each join tree node.

Figure 3.6 illustrates how the procedure `DP-Linear-1` works. In its first loop, it initializes the bottom row of join trees of size one. Then it computes the join trees joining exactly two relations. This is indicated by the next group of join trees. Since the figure leaves out commutativity, only one alternative join tree for every subset of size two is generated. This changes for subsets of size three. There, three alternative join trees are generated. Only the best join tree is retained. This is indicated by the ovals that encircle three join trees. Only this best join tree of size three is used to generate the final best join tree.

The short clarification after the algorithm already adumbrated that the order in which join trees are generated is not compulsory. The only necessary condition is the following.

> Let S be a subset of $\{R_1, \ldots, R_n\}$. Then, before a join tree for S can be generated, the join trees for all relevant subsets of S must already be available.

Note that this formulation is general enough to also capture the generation of bushy trees. It is, however, a little vague due to its reference to "relevance". For the different join tree classes, this term can be given a precise semantics. EX

Let us take a look at an alternative order to join tree generation. Assume that sets of relations are represented as bitvectors. A bitvector is nothing more than a base two integer. Successive increments of an integer/bitvector lead to different subsets. Further, the above condition is satisfied. We illustrate this by a small example. Assume that we have three relations R_1, R_2, R_3. The i-th bit from the right in a three-bit integer indicates the presence of R_i for $1 \leq i \leq 3$.

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

This observation leads to another formulation of our dynamic programming algorithm. For this algorithm, it is very convenient to use an array of size 2^n to represent `BestTree(`S`)` for subsets S of $\{R_1, \ldots, R_n\}$.

```
DP-Linear-2({R_1, ..., R_n})
Input: a set of relations to be joined
Output: an optimal left-deep (right-deep, zig-zag) join tree
for (i = 1; i <= n; ++i) {
  BestTree(1 << i) = R_i;
}
for (S = 1; S < 2^n; ++S) {
  if (BestTree(S) != NULL) continue;
  for all i ∈ S do {
    S' = S \ {i};
    CurrTree = CreateJoinTree(BestTree(S'), R_i);
    if (BestTree(S) == NULL || cost(BestTree(S)) > cost(CurrTree)) {
      BestTree(S) = CurrTree;
    }
  }
}
return BestTree(2^n - 1);
```

DP-Linear-2 differs from DP-Linear-1 not only in the order in which join trees are generated. Another difference is that it takes cross products into account.

From DP-Linear-2, it is easy to derive an algorithm that explores the space of bushy trees.

```
DP-Bushy({R_1, ..., R_n})
Input: a set of relations to be joined
Output: an optimal bushy join tree
for (i = 1; i <= n; ++i) {
  BestTree(1 << i) = R_i;
}
for (S = 1; S < 2^n; ++S) {
  if (BestTree(S) != NULL) continue;
  for all S_1 ⊂ S do {
    S_2 = S \ S_1;
    CurrTree = CreateJoinTree(BestTree(S_1), BestTree(S_2));
    if (BestTree(S) == NULL || cost(BestTree(S)) > cost(CurrTree)) {
      BestTree(S) = CurrTree;
    }
  }
}
return BestTree(2^n - 1);
```

This algorithm also takes cross products into account. The critical part is the generation of all subsets of S. Fortunately, Vance and Maier [860] provide a code fragment with which subset bitvector representations can be generated very efficiently. In C, this fragment looks as follows:

```
S1 = S & - S;
do {
  /* do something with subset S1 */
  S1 = S & (S1 - S);
} while (S1 != S);
```

S represents the input set. S_1 iterates through all subsets of S where S itself and the empty set are not considered. Analogously, all supersets an be generated as follows:

```
S1 = ~S & - ~S;
/* do something with first superset S1 */
while (S1) {
  S1 = ~S & (S1 - ~S)
  /* do something with superset S1
}
```

S represents the input set. S_1 iterates through all supersets of S including S itself.

Excursion Problem: exploiting orderings devastates the optimality principle. Example: ... XC ToDo

Excursion Pruning ... XC ToDo

Number of Entries to be stored in the dynamic programming table

If dynamic programming uses a static hash table, determining its size in advance is necessary as the search space sizes differ vastly for different query graphs. In general, for every connected subgraph of the query graph one entry must exist. Chains require far fewer entries than cliques. It would be helpful to have a small routine solving the following problem: given a query graph, how many connected subgraph are there? Unfortunatly, this problem is #-P hard as Sutner, Satyanarayana, and Suffel showed [825]. They build on results by Valiant [858] and Lichtenstein [540]. (For a definition of #P-hard see the book by Lewis and Papadimitriou [538] or the original paper by Valiant [857].)

However, for specific cases, these numbers can be given. If cross products are consideres, the number of join trees stored in the dynamic programming table is

$$2^n - 1$$

which is one for each non-empty subset of relations.

If we do not consider cross products, the number of entries in the dynamic programming table corresponds to the number of connected subgraphs of the query graph. For connected query graphs, we denote this by $\#\texttt{csg}$. For chains, cycles, stars, and

cliques with n nodes, we have

$$\#\texttt{csg}^{\text{chain}}(n) = \frac{n(n+1)}{2} \tag{3.2}$$
$$\#\texttt{csg}^{\text{cycle}}(n) = n^2 - n + 1 \tag{3.3}$$
$$\#\texttt{csg}^{\text{star}}(n) = 2^{n-1} + n - 1 \tag{3.4}$$
$$\#\texttt{csg}^{\text{clique}}(n) = 2^n - 1 \tag{3.5}$$

These equations can be derived from the following by summing over $k > 1$ where k gives the size of the connected subset:

$$\#\texttt{csg}^{\text{chain}}(n,k) = (n-k+1)$$
$$\#\texttt{csg}^{\text{cycle}}(n,k) = \begin{cases} 1 & n = k \\ n & \text{else} \end{cases}$$
$$\#\texttt{csg}^{\text{star}}(n,k) = \begin{cases} n & k = 1 \\ \binom{n-1}{k-1} & k > 1 \end{cases}$$
$$\#\texttt{csg}^{\text{clique}}(n,k) = \binom{n}{k}$$

Number of Join Trees Investigated

The number of join trees investigated by dynamic programming was extensively studied by Ono and Lohman [626, 627]. In order to estimate these numbers, we assume that `CreateJoinTree` produces a single join tree and hence counts as one although it may evaluate the costs for several join alternatives. We further do not count the initial join trees containing only a single relation.

Join Trees With Cartesian Product For the analysis of dynamic programming variants that do consider cross products, the notion of *join-pair* is helpful. Let S_1 and S_2 be subsets of the nodes (relations) of the query graph. We say (S_1, S_2) is a *join-pair*, if and only if

1. S_1 and S_2 are disjoint

If (S_1, S_2) is a join-pair, then (S_2, S_1) is a join pair. Further, if T_1 is a join tree for the relations in S_1 and T_2 is one for those in S_2, then we can construct two valid join trees $T_1 \bowtie T_2$ and $T_2 \bowtie T_1$ where the joins may be cross products. Hence, the number of join-pairs coincides with the search space a dynamic programming algorithm explores. In fact, the number of join-pairs is the minimum number of join trees any dynamic programming algorithm that considers cross products has to investigate.

If `CreateJoinTree` considers commutativity of joins, the number of calls to it is precisely expressed by the count of non-symmetric join-pairs. In other implementations `CreateJoinTree` might be called for all join-pairs and, thus, may not consider commutativity. The two formulas below only count non-symmetric join pairs.

The numbers of linear and bushy join trees with cartesian product is easiest to determine. They are independent of the query graph. For linear join trees, the number

of join trees investigated by dynamic programming is equal to the number of non-symmetric join-pairs which is

$$n2^{n-1} - \frac{n(n+1)}{2}$$

Dynamic programming investigates the following number of bushy trees if cross products are considered.

$$\frac{(3^n - 2^{n+1} + 1)}{2}$$

This is equal to the number of non-symmetric join-pairs.

Join Trees without Cross Products In this paragraph, we assume that the query graph is connected. For the analysis of dynamic programming variants that do not consider cross products, it is helpful to have the notion of a *csg-cmp-pair*. Let S_1 and S_2 be subsets of the nodes (relations) of the query graph. We say (S_1, S_2) is a *csg-cmp-pair*, if and only if

1. S_1 induces a connected subgraph of the query graph,
2. S_2 induces a connected subgraph of the query graph,
3. S_1 and S_2 are disjoint, and
4. there exists at least one edge connected a node in S_1 to a node in S_2.

If (S_1, S_2) is a csg-cmp-pair, then (S_2, S_1) is a valid csg-cmp-pair. Further, if T_1 is a join tree for the relations in S_1 and T_2 is one for those in S_2, then we can construct two valid join trees $T_1 \bowtie T_2$ and $T_2 \bowtie T_1$. Hence, the number of csg-cmp-pairs coincides with the search space a dynamic programming algorithm explores. In fact, the number of csg-cmp-pairs is the minimum number of join trees any dynamic programming algorithm that does not consider cross products has to investigate.

If `CreateJoinTree` considers commutativity of joins, the number of calls to it is precisely expressed by the count of non-symmetric csg-cmp-pairs. In other implementations `CreateJoinTree` might be called for all csg-cmp-pairs and, thus, may not consider commutativity.

Let us denote the number of non-symmetric csg-cmp-pairs by $\#\texttt{ccp}$. Then

$$\begin{aligned}
\#\texttt{ccp}^{\text{chain}}(n) &= \frac{1}{6}(n^3 - n) \\
\#\texttt{ccp}^{\text{cycle}}(n) &= (n^3 - 2n^2 + n)/2 \\
\#\texttt{ccp}^{\text{star}}(n) &= (n-1)2^{n-2} \\
\#\texttt{ccp}^{\text{clique}}(n) &= (3^n - 2^{n+1} + 1)/2
\end{aligned}$$

These numbers have to be multiplied by two if we want to count all csg-cmp-pairs.

If we do not consider composite inners, that is we restrict ourselves to left-deep join trees, then dynamic programming makes the following number of calls to `CreateJoinTree` for chain queries [627]:

$$(n-1)^2$$

The following table presents some results for the above formulas.

	without cross products			with cross products	
	chain		star	any query graph	
	linear	bushy	linear	linear	bushy
n	$(n-1)^2$	$(n^3-n)/6$	$(n-1)2^{n-2}$	$n2^{n-1}-n(n+1)/2$	$(3^n-2^{n+1}+1)/2$
2	1	1	1	1	1
3	4	4	4	6	6
4	9	10	12	22	25
5	16	20	32	65	90
6	25	35	80	171	301
7	36	56	192	420	966
8	49	84	448	988	3025
9	64	120	1024	2259	9330
10	81	165	2304	5065	28501

Compare this table with the actual sizes of the search spaces in Section 3.1.5.

The dynamic programming algorithms can be implemented very efficiently and often form the core of commercial plan generators. However, they have the disadvantage that no plan is generated if they run out of time or space since the search space they have to explore is too big. One possible remedy goes as follows. Assume that a dynamic programming algorithm is stopped in the middle of its way through its actual search space. Further assume that the largest plans generated so far involve k relations. Then the cheapest of the plans with k relations is completed by applying any heuristics (e.g. MinSel). The completed plan is then returned. In Section 3.4.5, we will see two alternative solutions. Another solution is presented in [476].

Generating Bushy Trees without Cross Products

We now discuss dynamic programming algorithms to generate bushy trees without cross products. For this section, we assume that the query graph is connected. We will present three algorithms. The first algorithm (`DPsize`) generates its plans in increasing size of subplans and, hence, is a generalization of `DP-Linear-1`. The second algorithm (`DPsub`) geneerates its plans by considering plans subsets as does `DP-Linear-2`. An analysis of these two algorithms reveals that both are far away from the lower bound presented in the previous sections. Thus, a third algorithm (`DPccp`) which reaches this lower bound is presented. The results of this section are taken from [601, 600].

Size-based enumeration: `DPsize` In general, dynamic programming generates solutions for a larger problem in a bottom-up fashion by combining solutions for smaller problems. Taking this description literally, we can construct optimal plans of size n by joining plans P_1 and P_2 of size k and $n-k$. We just have to take care that (1) the sets of relations contained in P_1 and P_2 do not overlap, and (2) there is a join predicate connecting a relation P_1 with a relation in P_2. After this remark, we are prepared to understand the pseudocode for algorithm `DPsize` (see Fig. 3.7). A table `BestPlan` associates with each set of relations the best plan found so far. The algorithm starts

```
DPsize
Input: a connected query graph with relations R = {R_0, ..., R_{n-1}}
Output: an optimal bushy join tree without cross products
for all R_i ∈ R {
    BestPlan({R_i}) = R_i;
}
for all 1 < s ≤ n ascending // size of plan
for all 1 ≤ s_1 < s { // size of left subplan
    s_2 = s − s_1; // size of right subplan
    for all  S_1 ⊂ R : |S_1| = s_1
             S_2 ⊂ R : |S_2| = s_2 {
        ++InnerCounter;
        if (∅ ≠ S_1 ∩ S_2) continue;
        if not (S_1 connected to S_2) continue;
        ++CsgCmpPairCounter;
        p_1=BestPlan(S_1);
        p_2=BestPlan(S_2);
        CurrPlan = CreateJoinTree(p_1, p_2);
        if (cost(BestPlan(S_1 ∪ S_2)) > cost(CurrPlan)) {
            BestPlan(S_1 ∪ S_2) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R_0, ..., R_{n-1}});
```

Figure 3.7: Algorithm DPsize

by initializing this table with plans of size one, i.e. single relations. After that, it constructs plans of increasing size (loop over s). Thereby, the first size considered is two, since plans of size one have already been constructed. Every plan joining n relations can be constructed by joining a plan containing s_1 relations with a plan containing s_2 relations. Thereby, $s_i > 0$ and $s_1 + s_2 = n$ must hold. Thus, the pseudocode loops over s_1 and sets s_2 accordingly. Since for every possible size there exist many plans, two more loops are necessary in order to loop over the plans of sizes s_1 and s_2. Then, conditions (1) and (2) from above are tested. Only if their outcome is positive, we consider joining the plans p_1 and p_2. The result is a plan `CurrPlan`. Let S be the relations contained in `CurrPlan`. If `BestPlan` does not contain a plan for the relations in S or the one it contains is more expensive than `CurrPlan`, we register `CurrPlan` with `BestPlan`.

The algorithm `DPsize` can be made more efficient in case of $s_1 = s_2$. The algorithm as stated cycles through all plans p_1 joining s_1 relations. For each such plan, all plans p_2 of size s_2 are tested. Assume that plans of equal size are represented as a linked list. If $s_1 = s_2$, then it is possible to iterate through the list for retrieving all plans p_1. For p_2 we consider the plans succeeding p_1 in the list. Thus, the complexity can be decreased from $s_1 * s_2$ to $s_1 * s_2/2$ The following formulas are valid only for the variant of `DPsize` where this optimization has been incorporated (see [600] for

details).

If the counter `InnerCounter` is initialized with zero at the beginning of the algorithm `DPsize`, then we are able to derive analytically its value after `DPsize` terminates. Since this value of the inner counter depends on the query graph, we have to distinguish several cases. For chain, cycle, star, and clique queries, we denote by $I^{\text{chain}}_{\text{DPsize}}$, $I^{\text{cycle}}_{\text{DPsize}}$, $I^{\text{star}}_{\text{DPsize}}$, and $I^{\text{clique}}_{\text{DPsize}}$ the value of `InnerCounter` after termination of algorithm `DPsize`.

For chain queries, we then have: $I^{\text{chain}}_{\text{DPsize}}(n) =$

$$\begin{cases} 1/48(5n^4 + 6n^3 - 14n^2 - 12n) & n \text{ even} \\ 1/48(5n^4 + 6n^3 - 14n^2 - 6n + 11) & n \text{ odd} \end{cases}$$

For cycle queries, we have: $I^{\text{cycle}}_{\text{DPsize}}(n) =$

$$\begin{cases} \frac{1}{4}(n^4 - n^3 - n^2) & n \text{ even} \\ \frac{1}{4}(n^4 - n^3 - n^2 + n) & n \text{ odd} \end{cases}$$

For star queries, we have: $I^{\text{star}}_{\text{DPsize}}(n) =$

$$\begin{cases} 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + q(n) & n \text{ even} \\ 2^{2n-4} - 1/4\binom{2(n-1)}{n-1} + 1/4\binom{n-1}{(n-1)/2} + q(n) & n \text{ odd} \end{cases}$$

with $q(n) = n2^{n-1} - 5 * 2^{n-3} + 1/2(n^2 - 5n + 4)$. For clique queries, we have: $I^{\text{clique}}_{\text{DPsize}}(n) =$

$$\begin{cases} 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} - 1/4\binom{n}{n/2} + 1 & n \text{ even} \\ 2^{2n-2} - 5 * 2^{n-2} + 1/4\binom{2n}{n} + 1 & n \text{ odd} \end{cases}$$

Note that $\binom{2n}{n}$ is in the order of $\Theta(4^n/\sqrt{n})$.

Proofs of the above formulas as well as implementation details for the algorithm `DPsize` can be found in [600].

Subset-Driven Enumeration: `DPsub` Fig. 3.8 presents the pseudocode for the algorithm `DPsub`. The algorithm first initializes the table `BestPlan` with all possible plans containing a single relation. Then, the main loop starts. It iterates over all possible non-empty subsets of $\{R_0, \ldots, R_{n-1}\}$ and constructs the best possible plan for each of them. The enumeration makes use of a bitvector representation of sets: The integer i induces the current subset S with its binary representation. Taken as bitvectors, the integers in the range from 1 to $2^n - 1$ exactly represent the set of all non-empty subsets of $\{R_0, \ldots, R_{n-1}\}$, including the set itself. Further, by starting with 1 and incrementing by 1, the enumeration order is valid for dynamic programming: for every subset, all its subsets are generated before the subset itself.

This enumeration is very fast, since increment by one is a very fast operation. However, the relations contained in S may not induce a connected subgraph of the query graph. Therefore, we must test for connectedness. The goal of the next loop over all subsets of S is to find the best plan joining all the relations in S. Therefore, S_1 ranges over all non-empty, strict subsets of S. This can be done very efficiently by

```
DPsub
Input: a connected query graph with relations R = {R_0, ..., R_{n-1}}
Output: an optimal bushy join tree
for all R_i ∈ R {
    BestPlan({R_i}) = R_i;
}
for 1 ≤ i < 2^n − 1 ascending {
    S = {R_j ∈ R|(⌊i/2^j⌋ mod 2) = 1}
    if not (connected S) continue;    // *
    for all S_1 ⊂ S, S_1 ≠ ∅ do {
        ++InnerCounter;
        S_2 = S \ S_1;
        if (S_2 = ∅) continue;
        if not (connected S_1) continue;
        if not (connected S_2) continue;
        if not (S_1 connected to S_2) continue;
        ++CsgCmpPairCounter;
        p_1 = BestPlan(S_1);
        p_2 = BestPlan(S_2);
        CurrPlan = CreateJoinTree(p_1, p_2);
        if (cost(BestPlan(S)) > cost(CurrPlan)) {
            BestPlan(S) = CurrPlan;
        }
    }
}
OnoLohmanCounter = CsgCmpPairCounter / 2;
return BestPlan({R_0, ..., R_{n-1}});
```

Figure 3.8: Algorithm DPsub

applying the code snippet of Vance and Maier [859, 860]. Then, the subset of relations contained in S but not in S_1 is assigned to S_2. Clearly, S_1 and S_2 are disjoint. Hence, only connectedness tests have to be performed. Since we want to avoid cross products, S_1 and S_2 both must induce connected subgraphs of the query graph, and there must be a join predicate between a relation in S_1 and one in S_2. If these conditions are fulfilled, we can construct a plan `CurrPlan` by joining the plans associated with S_1 and S_2. If `BestPlan` does not contain a plan for the relations in S or the one it contains is more expensive than `CurrPlan`, we register `CurrPlan` with `BestPlan`.

For chain, cycle, star, and clique queries, we denote by $I^{\text{chain}}_{\text{DPsub}}$, $I^{\text{cycle}}_{\text{DPsub}}$, $I^{\text{star}}_{\text{DPsub}}$, and $I^{\text{clique}}_{\text{DPsub}}$ the value of `InnerCounter` after termination of algorithm `DPsub`.

For chains, we have

$$I^{\text{chain}}_{\text{DPsub}}(n) = 2^{n+2} - n^n - 3n - 4 \tag{3.6}$$

For cycles, we have

$$I^{\text{cycle}}_{\text{DPsub}}(n) = n2^n + 2^n - 2n^2 - 2 \tag{3.7}$$

	Chain			Cycle		
n	#ccp	DPsub	DPsize	#ccp	DPsub	DPsize
2	1	2	1	1	2	1
5	20	84	73	40	140	120
10	165	3962	1135	405	11062	2225
15	560	130798	5628	1470	523836	11760
20	1330	4193840	17545	3610	22019294	37900
	Star			Clique		
n	#ccp	DPsub	DPsize	#ccp	DPsub	DPsize
2	1	2	1	1	2	1
5	32	130	110	90	180	280
10	2304	38342	57888	28501	57002	306991
15	114688	9533170	57305929	7141686	14283372	307173877
20	4980736	2323474358	59892991338	1742343625	3484687250	309338182241

Figure 3.9: Size of the search space for different graph structures

For stars, we have

$$I^{\text{star}}_{\text{DPsub}}(n) = 2 * 3^{n-1} - 2^n \tag{3.8}$$

For cliques, we have

$$I^{\text{clique}}_{\text{DPsub}}(n) = 3^n - 2^{n+1} + 1 \tag{3.9}$$

The number of failures for the additional check can easily be calculated as $2^n - \#\texttt{csg}(n) - 1$.

Sample numbers Fig. 3.9 contains tables with values produced by our formulas for input query graph sizes between 2 and 20. For different kinds of query graphs, it shows the number of csg-cmp-pairs (#ccp). and the values for the inner counter after termination of DPsize and DPsub applied to the different query graphs.

Looking at these numbers, we observe the following:

- For chain and cycle queries, the DPsize soon becomes much faster than DPsub.
- For star and clique queries, the DPsub soon becomes much faster than DPsize.
- Except for clique queries, the number of csg-cmp-pairs is orders of magnitude less than the value of *InnerCounter* for all DP-variants.

From the latter observation we can conclude that in almost all cases the tests performed by both algorithms in their innermost loop fail. Both algorithms are far away from the theoretical lower bound given by #ccp. This conclusion motivates us to derive a new algorithm whose *InnerCounter* value is equal to the number of csg-cmp-pairs.

Csg-cmp-pair enumeration-based algorithm: `DPccp` The algorithm `DPsub` solves the join ordering problem for a given subset S of relations by considering all pairs of disjoint subproblems which were already solved. Since the enumeration of subsets is very fast, this is a very efficient strategy if the search space is dense, e.g. for clique queries. However, if the search space is sparse, e.g. for chain queries, the `DPsub` algorithm considers many subproblems which are not connected and, therefore, are not relevant for the solution, i.e. the tests in the innermost loop fail for the majority of cases. The main idea of our algorithm `DPccp` is that it only considers pairs of connected subproblems. More precisely, the algorithm considers exactly the csg-cmp-pairs of a graph.

Thus, our goal is to efficiently enumerate all csg-cmp-pairs (S_1, S_2). Clearly, we want to enumerate every pair once and only once. Further, the enumeration must be performed in an order valid for dynamic programming. That is, whenever a pair (S_1, S_2) is generated, all non-empty subsets of S_1 and S_2 must have been generated before as a component of a pair. The last requirement is that the overhead for generating a single csg-cmp-pair must be constant or at most linear. This condition is necessary in order to beat `DPsize` and `DPsub`.

`DPccp`
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree
for all $R_i \in R$) {
    BestPlan($\{R_i\}$) = R_i;
}
for all csg-cmp-pairs (S_1, S_2), $S = S_1 \cup S_2$ {
    ++InnerCounter;
    ++OnoLohmanCounter;
    p_1 = BestPlan(S_1);
    p_2 = BestPlan(S_2);
    CurrPlan = CreateJoinTree(p_1, p_2);
    if (cost(BestPlan(S)) $>$ cost(CurrPlan)) {
        BestPlan(S) = CurrPlan;
    }
    CurrPlan = CreateJoinTree(p_2, p_1);
    if (cost(BestPlan(S)) $>$ cost(CurrPlan)) {
        BestPlan(S) = CurrPlan;
    }
}
CsgCmpPairCounter = 2 * OnoLohmanCounter;
return BestPlan($\{R_0, \ldots, R_{n-1}\}$);

Figure 3.10: Algorithm `DPccp`

If we meet all these requirements, the algorithm `DPccp` is easily specified: iterate over all csg-cmp-pairs (S_1, S_2) and consider joining the best plans associated with them. Figure 3.10 shows the pseudocode. The first steps of an example enumeration are shown in Figure 3.11. Thick lines mark the connected subsets while thin lines mark

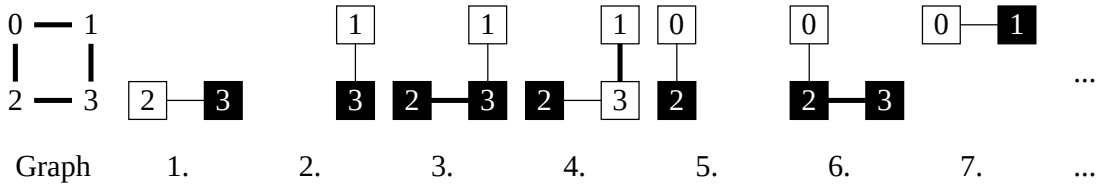


Figure 3.11: Enumeration Example for `DPccp`

possible join edges. Note that the algorithm explicitly exploits join commutativity. This is due to our enumeration algorithm developed below. If (S_1, S_2) is a csg-cmp-pair, then either (S_1, S_2) or (S_2, S_1) will be generated, but never both of them. An alternative is to modify `CreateJoinTree` to take care of commutativity.

We proceed as follows. Next we discuss an algorithm enumerating non-empty connected subsets S_1 of $\{R_0, \ldots, R_{n-1}\}$. Then, we show how to enumerate the complements S_2 such that (S_1, S_2) is a csg-cmp-pair.

Let us start the exposition by fixing some notations. Let $G = (V, E)$ be an undirected graph. For a node $v \in V$ define the *neighborhood* $\mathcal{N}(v)$ of v as $\mathcal{N}(v) := \{v' | (v, v') \in E\}$. For a subset $S \subseteq V$ of V we define the *neighborhood* of S as $\mathcal{N}(S) := \cup_{v \in S} \mathcal{N}(v) \setminus S$. The neighborhood of a set of nodes thus consists of all nodes reachable by a single edge. Note that for all $S, S' \subset V$ we have $\mathcal{N}(S \cup S') = (\mathcal{N}(S) \cup \mathcal{N}(S')) \setminus (S \cup S')$. This allows for an efficient bottom-up calculation of neighborhoods.

The following statement gives a hint on how to construct an enumeration procedure for connected subsets. Let S be a connected subset of an undirected graph G and S' be any subset of $\mathcal{N}(S)$. Then $S \cup S'$ is connected. As a consequence, a connected subset can be enlarged by adding any subset of its neighborhood.

We could generate all connected subsets as follows. For every node $v_i \in V$ we perform the following enumeration steps: First, we emit $\{v_i\}$ as a connected subset. Then, we expand $\{v_i\}$ by calling a routine that extends a given connected set to bigger connected sets. Let the routine be called with some connected set S. It then calculates the neighborhood $\mathcal{N}(S)$. For every non-empty subset $N \subseteq \mathcal{N}(S)$, it emits $S' = S \cup N$ as a further connected subset and recursively calls itself with S'. The problem with this routine is that it produces duplicates.

This is the point where the breadth-first numbering comes into play. Let $V = \{v_0, \ldots, v_{n-1}\}$, where the indices are consistent with a breadth-first numbering produced by a breadth-first search starting at node v_0 [206]. The idea is to use the numbering to define an enumeration order: In order to avoid duplicates, the algorithm enumerates connected subgraphs for every node v_i, but restricts them to contain no v_j with $j < i$. Using the definition $\mathcal{B}_i = \{v_j | j \leq i\}$, the pseudocode looks as follows:

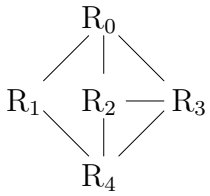


Figure 3.12: Sample graph to illustrate `EnumerateCsgRec`

`EnumerateCsg`
Input: a connected query graph $G = (V, E)$
Precondition: nodes in V are numbered according to a breadth-first search
Output: emits all subsets of V inducing a connected subgraph of G
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

Let us consider an example. Figure 3.12 contains a query graph whose nodes are numbered in a breadth-first fashion. The calls to `EnumerateCsgRec` are contained in the table in Figure 3.13. In this table, S and X are the arguments of `EnumerateCsgRec`. N is the local variable after its initialization. The column `emit`/S contains the connected subset emitted, which then becomes the argument of the recursive call to `EnumerateCsgRec` (labelled by $\rightarrow$). Since listing all calls is too lengthy, only a subset of the calls is listed.

Generating the connected subsets is an important first step but clearly not sufficient: we have to generate all csg-cmp-pairs. The basic idea to do so is as follows. Algorithm `EnumerateCsg` is used to create the first component S_1 of every csg-cmp-pair. Then, for each such S_1, we generate all its complement components S_2. This can be done by calling `EnumerateCsgRec` with the correct parameters. Remember that we have to generate every csg-cmp-pair once and only once.

To achieve this, we use a similar technique as for connected subsets, using the breadth-first numbering to define an enumeration order: we consider only sets S_2 in the complement of S_1 (with (S_1, S_2) being a csg-cmp-pair) such that S_2 contains only v_j with j larger than any i with $v_i \in S_1$. This avoids the generation of duplicates.

We need some definitions to state the actual algorithm. Let $S_1 \subseteq V$ be a non-empty subset of V. Then, we define $\min(S_1) := \min(\{i | v_i \in S_1\})$. This is used to

EnumerateCsgRec			
S	X	N	emit/S
$\{4\}$	$\{0,1,2,3,4\}$	$\emptyset$	
$\{3\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{3,4\}$
$\{2\}$	$\{0,1,2\}$	$\{3,4\}$	
			$\{2,3\}$
			$\{2,4\}$
			$\{2,3,4\}$
$\{1\}$	$\{0,1\}$	$\{4\}$	
			$\{1,4\}$
$\rightarrow \{1,4\}$	$\{0,1,4\}$	$\{2,3\}$	
			$\{1,2,4\}$
			$\{1,3,4\}$
			$\{1,2,3,4\}$
$\{0\}$	$\{0\}$	$\{1,2,3\}$	
			$\{0,1\}$
			$\{0,2\}$
			$\{0,3\}$
			$\{0,1,2\}$
			$\{0,1,3\}$
			$\{0,2,3\}$
			$\{0,1,2,3\}$
$\rightarrow \{0,1\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{0,1,4\}$
$\rightarrow \{0,2\}$	$\{0,1,2,3\}$	$\{4\}$	
			$\{0,2,4\}$

Figure 3.13: Call sequence for Figure 3.12

extract the starting node from which S_1 was constructed (see Lemma **??**). Let $W \subset V$ be a non-empty subset of V. Then, we define $\mathcal{B}_i(W) := \{v_j | v_j \in W, j \leq i\}$. Using this notation, the algorithm to construct all S_2 for a given S_1 such that (S_1, S_2) is a csg-cmp-pair looks as follows:

`EnumerateCmp`
Input: a connected query graph $G = (V, E)$, a connected subset S_1
Precondition: nodes in V are numbered according to a breadth-first search
Output: emits all complements S_2 for S_1 such that (S_1, S_2) is a csg-cmp-pair
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup \mathcal{B}_i(N)$);
}

Algorithm `EnumerateCmp` considers all neighbors of S_1. First, they are used to determine those S_2 that contain only a single node. Then, for each neighbor of S_1, it recursively calls `EnumerateCsgRec` to create those S_2 that contain more than a single node. Note that here both nodes concerning the enumeration of S_1 ($\mathcal{B}_{\min(S_1)} \cup S_1$) and nodes concerning the enumeration of S_2 (N) have to be considered in order to guarantee a correct enumeration. Otherwise the combined algorithm would emit (commutative) duplicates.

Let us consider an example for algorithm `EnumerateCmp`. The underlying graph is again the one shown in Fig. 3.12. Assume `EnumerateCmp` is called with $S_1 = \{R_1\}$. In the first statement, the set $\{R_0, R_1\}$ is assigned to X. Then, the neighborhood is calculated. This results in

$$N = \{R_0, R_4\} \setminus \{R_0, R_1\} = \{R_4\}.$$

Hence, $\{R_4\}$ is emitted and together with $\{R_1\}$, it forms the csg-cmp-pair $(\{R_1\}, \{R_4\})$. Then, the recursive call to `EnumerateCsgRec` follows with arguments G, $\{R_4\}$, and $\{R_0, R_1, R_4\}$. Subsequent `EnumerateCsgRec` generates the connected sets $\{R_2, R_4\}$, $\{R_3, R_4\}$, and $\{R_2, R_3, R_4\}$, giving three more csg-cmp-pairs.

3.2.5 Memoization

Whereas dynamic programming constructs the join trees iteratively from small trees to larger trees, i.e. works bottom up, memoization works recursively. For a given set of relations S, it produces the best join tree for S by recursively calling itself for every subset S_1 of S and considering all join trees between S_1 and its complement S_2. The best alternative is memoized (hence the name). The reason is that two (even different) (sub-) sets of all relations may very well have the common subsets. For example, $\{R_1, R_2, R_3, R_4, R_5\}$ and $\{R_2, R_3, R_4, R_5, R_6\}$ have the common subset $\{R_2, R_3, R_4, R_5\}$. In order to avoid duplicate work, memoization is essential.

In the following variant of memoization, we explore the search space of all bushy trees and consider cross products. We split the functionality across two functions. The first one initializes the `BestTree` data structure with single relation join trees for R_i and then calls the second one. The second one is the core memoization procedure which calls itself recursively. EX

```
MemoizationJoinOrdering(R)
Input:  a set of relations R
Output:  an optimal join tree for R
for (i = 1; i <= n; ++i) {
  BestTree({R_i}) = R_i;
}
return MemoizationJoinOrderingSub(R);

MemoizationJoinOrderingSub(S)
Input:  a (sub-) set of relations S
Output:  an optimal join tree for S
```

```
if(NULL == BestTree(S)) {
  for all S_1 ⊂ S do {
    S_2 = S \ S_1;
    CurrTree = CreateJoinTree(MemoizationJoinOrderingSub(S_1), Memoization
    if (BestTree(S) == NULL || cost(BestTree(S)) > cost(CurrTree)) {
      BestTree(S) = CurrTree;
    }
  }
}
return BestTree(S);
```

ToDo? Again, pruning techniques can help to speed up plan generation [767].

3.2.6 Join Ordering by Generating Permutations

For any set of cost functions, we can directly generate permutations. Generating all permutations is clearly too expensive for more than a couple of relations. However, we can safely neglect some of them. Consider the join sequence $R_1R_2R_3R_4$. If we know that $R_1R_3R_2$ is cheaper than $R_1R_2R_3$, we do not have to consider $R_1R_2R_3R_4$. The idea of the following algorithm is to construct permutations by successively adding relations. Thereby, an extended sequence is only explored if exchanging the last two relations does not result in a cheaper sequence.

```
ConstructPermutations(Query Specification)
Input: query specification for relations {R_1, ..., R_n}
Output: optimal left-deep tree
BestPermutation = NULL;
Prefix = ε;
Rest = {R_1, ..., R_n};
ConstructPermutationsSub(Prefix, Rest);
return BestPermutation

ConstructPermutationsSub(Prefix, Rest)
Input: a prefix of a permutation and the relations to be added (Rest)
Ouput: none, side-effect on BestPermutation
if (Rest == ∅) {
  if (BestPermutation == NULL || cost(Prefix) < cost(BestPermutation)) {
    BestPermutation = Prefix;
  }
  return
}
foreach (R_i, R_j ∈ Rest) {
  if (cost(Prefix ∘ ⟨R_i, R_j⟩) ≤ cost(Prefix ∘ ⟨R_j, R_i⟩)) {
    ConstructPermutationsSub(Prefix ∘ ⟨R_i⟩, Rest \ {R_i});
  }
  if (cost(Prefix ∘ ⟨R_j, R_i⟩) ≤ cost(Prefix ∘ ⟨R_i, R_j⟩)) {
```

```
    ConstructPermutationsSub(Prefix ∘ ⟨R_j⟩, Rest \ {R_j});
  }
}
return
```

The algorithm can be made more efficient, if the `foreach` loop considers only a single relation and performs the swap test with this relation and the last relation occurring in `Prefix`.

The algorithm has two main advantages over dynamic programming and memoization. The first advantage is that it needs only linear space opposed to exponential space for the two mentioned alternatives. The other main advantage over dynamic programming is that it generates join trees early, whereas with dynamic programming we only generate a plan after the whole search space has been explored. Thus, if the query contains too many joins—that is, the search space cannot be fully explored in reasonable time and space—dynamic programming will not generate any plan at all. If stopped, `ConstructPermutations` will not necessarily compute the best plan, but still some plans have been investigated. This allows us to stop it after some time limit has exceeded. The time limit itself can be fixed, like 100 ms, or variable, like 5% of the execution time of the best plan found so far.

The predicates in the **if** statement can be made more efficient if a (local) ranking function is available. Further speed-up of the algorithm can be achieved if additionally the idea of memoization is applied (of course, this jeopardizes the small memory footprint).

The following variant might be interesting if one is willing to go from linear space consumption to quadratic space consumption. The original algorithm is then started n times, once for each relation as a starting relation. The n different instantiations then have to run interleaved. This variant reduces the dependency on the starting relation.

Worst Case Analysis ToDo/EX

Pruning/memoization/propagation ToDo/EX

3.2.7 A Dynamic Programming based Heuristics for Chain Queries

In Section 3.1.6, we saw that the complexity of producing optimal left-deep trees possibly containing cross products for chain queries is an open problem. However, the case does not seem to be hopeless. In fact, Scheufele and Moerkotte present two algorithms [733, 735] for this problem. For one algorithm, it can be proven that it has polynomial runtime, for the other, it can be proven that it produces the optimal join tree. However, for none of them both could be proven so far.

Basic Definitions and Lemmata

An instance of the *join-ordering problem for chain queries* (or a *chain query* for short) is fully described by the following parameters. First, n relations $R_1, \ldots, R_n$ are given. The size of relation R_i $(1 \leq i \leq n)$ is denoted by $|R_i|$ or n_{R_i}. Second, the query graph G on the set of relations $R_1, \ldots, R_n$ must be a chain. That is, its edges are $\{(R_i, R_{i+1}) \,|\, 1 \leq i < n\}$:

$$R_1 — R_2 — \ldots — R_n$$

For every edge (R_i, R_{i+1}), there is an associated selectivity $f_{i,i+1} = |R_i \bowtie R_{i+1}|/|R_i \times R_{i+1}|$. We define all other selectivities $f_{i,j} = 1$ for $|i - j| \neq 1$. They correspond to cross products.

In this section we consider only left-deep processing trees. However, we allow them to contain cross products. Hence, any permutation is a valid join tree. There is a unique correspondence not only between left-deep join trees but also between consecutive parts of a permutation and segments of a left-deep tree. Furthermore, if a segment of a left-deep tree does not contain cross products, it uniquely corresponds to a consecutive part of the chain in the query graph. In this case, we also speak of (sub)chains or connected (sub)sequences. We say that two relations R_i and R_j are *connected* if they are adjacent in G; more generally, two sequences s and t are connected if there exist relations R_i in s and R_j in t such that R_i and R_j are connected. A sequence of relations s is connected if for all subsequences s_1 and s_2 satisfying $s = s_1 s_2$ it holds that s_1 is connected to s_2.

Given a chain query, we ask for a permutation $s = r_1 \ldots r_n$ of the n relations (i.e. there is a permutation π such that $r_i = R_{\pi(i)}$ for $1 \leq i \leq n$) that produces minimal costs under the cost function C_{out}.

Remember that the dynamic programming approach considers $n2^{n-1} - n(n+1)/2$ alternatives for left-deep processing trees with cross products—independently of the query graph and the cost function. The question arises whether it is possible to lower the complexity in case of simple chain queries.

The IKKBZ algorithm solves the join ordering problem for tree queries by decomposing the problem into polynomially many subproblems which are subject to tree-like precedence constraints. The precedence constraints ensure that the cost functions of the subproblems now have the ASI property. The remaining problem is to optimize the constrained subproblems under the simpler cost function. Unfortunately, this approach does not work in our case, since no such decomposition seems to exist.

Let us introduce some notions used for the algorithms. We have to generalize the rank used in the IKKBZ algorithm to *relativized ranks*. We start by relativizing the cost function. The costs of a sequence s *relative* to a sequence u are defined as

$$\begin{aligned}
C_u(\epsilon) &:= 0 \\
C_u(R_i) &:= 0 \text{ if } u = \epsilon \\
C_u(R_i) &:= (\prod_{R_j <_u R_i R_i} f_{j,i}) n_i \quad \text{if } u \neq \epsilon \\
C_u(s_1 s_2) &:= C_u(s_1) + T_u(s_1) * C_{us_1}(s_2)
\end{aligned}$$

with

$$\begin{aligned}
T_u(\epsilon) &:= 1 \\
T_u(s) &:= \prod_{R_i \in s} (\prod_{R_j <_u s R_i} f_{j,i}) * n_i
\end{aligned}$$

Here, $R_i <_s R_j$ is true if and only if R_i appears before R_j in s. As usual, empty products evaluate to 1. Several things should be noted. First, $C_{us}(t) = C_u(t)$ holds if there is no connection between relations in s and t. Second, $T_\epsilon(R_i) = |R_i|$ and $T_\epsilon(s) = |s|$. That is, T_u generalizes the size of a single relation or of a sequence of relations. Third, note that $C_u(\epsilon) = 0$ for all u but $C_\epsilon(s) = 0$ only if s does not contain more than one relation. The special case that $C_\epsilon(R) = 0$ for a single relation R causes some problems in the homogeneity of definitions and proofs. Hence, we abandon this case from all definitions and lemmata of this section. This will not be repeated in every definition and lemma, but will implicitly be assumed. Further, the two algorithms will be presented in two versions. The first version is simpler and relies on a modified cost function C', and only the second version will apply to the original cost function C. As we will see, C' differs from C in exactly the problematic case in which it is defined as $C'_u(R_i) := |R_i|$. Now, $C'_\epsilon(s) = 0$ holds if and only if $s = \epsilon$ holds. Within subsequent definitions and lemmata, C can also be replaced by C' without changing their validity. Last, we abbreviate C_ϵ by C for convenience.

Example 1: Consider a chain query involving the relations R_1, R_2, R_3. The parameters are $|R_1| = 1, |R_2| = 100, |R_3| = 10$ and $f_{1,2} = f_{2,3} = 0.9$. The expected size of the query result is independent of the ordering of the relations. Hence, we have

$$T(R_1R_2R_3) = \cdots = T(R_3R_2R_1) = 100 * 10 * 1 * .9 * .9 = 810.$$

There are 6 possible orderings of the relations with the following costs:

$$\begin{aligned}
C(R_1R_2R_3) &= 1 * 100 * 0.9 + 1 * 100 * 10 * 0.9 * 0.9 = 900 \\
C(R_1R_3R_2) &= 1 * 10 + 1 * 10 * 100 * 0.9 * 0.9 = 820 \\
C(R_2R_3R_1) &= 100 * 10 * 0.9 + 100 * 10 * 1 * 0.9 * 0.9 = 1710 \\
C(R_2R_1R_3) &= C(R_1R_2R_3) \\
C(R_3R_1R_2) &= C(R_1R_3R_2) \\
C(R_3R_2R_1) &= C(R_2R_3R_1)
\end{aligned}$$

Note that the cost function is invariant with respect to the order of the first two relations. The minimum over all costs is 820, and the corresponding optimal join ordering is $R_1R_3R_2$.

□

Using the relativized cost function, we can define the relativized rank.

Definition 3.2.8 (rank) *The* rank *of a sequence s relative to a non-empty sequence u is given by*

$$rank_u(s) \quad := \quad \frac{T_u(s) - 1}{C_u(s)}$$

In the special case that s consists of a single relation R_i, the intuition behind the *rank* function becomes transparent. Let f_i be the product of the selectivities between relations in u and R_i. Then $rank_u(R_i) = \frac{f_i|R_i|-1}{f_i|R_i|}$. Hence, the *rank* becomes a function of the form $f(x) = \frac{x-1}{x}$. This function is monotonously increasing in x for $x > 0$. The argument to the function $f(x)$ is (for the computation of the size of a

single relation R_i) $f_i|R_i|$. But this is the factor by which the next intermediate result will increase (or decrease). Since we sum up intermediate results, this is an essential number. Furthermore, it follows from the monotonicity of $f(x)$ that $rank_u(R_i) \leq rank_u(R_j)$ if and only if $f_i|R_i| \leq f_j|R_j|$ where f_j is the product of all selectivities between R_j and relations in u.

Example 1 (cont'd): Supposing the query given in Example 1, the optimal sequence $R_1R_3R_2$ gives rise to the following ranks.

$$rank_{R_1}(R_2) = \frac{T_{R_1}(R_2)-1}{C_{R_1}(R_2)} = \frac{100*0.9-1}{100*0.9} \approx 0.9888$$
$$rank_{R_1}(R_3) = \frac{T_{R_1}(R_3)-1}{C_{R_1}(R_3)} = \frac{10*1.0-1}{10*1.0} = 0.9$$
$$rank_{R_1R_3}(R_2) = \frac{T_{R_1R_3}(R_2)-1}{C_{R_1R_3}(R_2)} = \frac{100*0.9*0.9-1}{100*0.9*0.9} \approx 0.9877$$

Hence, within the optimal sequence, the relation with the smallest rank (here R_3, since $rank_{R_1}(R_3) < rank_{R_1}(R_2)$) is preferred. As the next lemma will show, this is no accident.

□

Using the rank function, the following lemma can be proved.

Lemma 3.2.9 *For sequences*

$$\begin{aligned} S &= r_1 \cdots r_{k-1} r_k r_{k+1} r_{k+2} \cdots r_n \\ S' &= r_1 \cdots r_{k-1} r_{k+1} r_k r_{k+2} \cdots r_n \end{aligned}$$

the following holds:

$$C(S) \leq C(S') \Leftrightarrow rank_u(r_k) \leq rank_u(r_{k+1})$$

where $u = r_1 \cdots r_{k-1}$. Equality only holds if it holds on both sides.

Example 1 (cont'd): Since the ranks of the relations in Example 1 are ordered with ascending ranks, Lemma 3.2.9 states that, whenever we exchange two adjacent relations, the costs cannot decrease. In fact, we observe that $C(R_1R_3R_2) \leq C(R_1R_2R_3)$.
□

An analogous lemma still holds for two unconnected subchains:

Lemma 3.2.10 *Let u, x and y be three subchains where x and y are not interconnected. Then we have:*

$$C(uxy) \leq C(uyx) \Leftrightarrow rank_u(x) \leq rank_u(y)$$

Equality only holds if it holds on both sides.

Next, we define the notion of a *contradictory chain*, which will be essential to the algorithms. The subsequent lemmata will allow us to cut down the search space to be explored by any optimization algorithm.

Definition 3.2.11 (contradictory pair of subchains) *Let u, x, y be nonempty sequences. We call (x, y) a* contradictory pair of subchains *if and only if*

$$C_u(xy) \leq C_u(yx) \quad \wedge \quad rank_u(x) > rank_{ux}(y)$$

A special case occurs when x and y are single relations. Then the above condition simplifies to

$$rank_{ux}(y) < rank_u(x) \leq rank_u(y)$$

To explain the intuition behind the definition of contradictory subchains, we need another example.

Example 2: Suppose a chain query involving R_1, R_2, R_3 is given. The relation sizes are $|R_1| = 1, |R_2| = |R_3| = 10$ and the selectivities are $f_{1,2} = 0.5, f_{2,3} = 0.2$. Consider the sequences $R_1R_2R_3$ and $R_1R_3R_2$, which differ in the order of the last two relations. We have

$$\begin{aligned} rank_{R_1}(R_2) &= 0.8 \\ rank_{R_1R_2}(R_3) &= 0.0 \\ rank_{R_1}(R_3) &= 0.9 \\ rank_{R_1R_3}(R_2) &= 0.5 \end{aligned}$$

and

$$\begin{aligned} C(R_1R_2R_3) &= 15 \\ C(R_1R_3R_2) &= 20 \end{aligned}$$

Hence,

$$\begin{aligned} rank_{R_1}(R_2) &> rank_{R_1R_2}(R_3) \\ rank_{R_1}(R_3) &> rank_{R_1R_3}(R_2) \\ C(R_1R_2R_3) &< C(R_1R_3R_2) \end{aligned}$$

and (R_2, R_3) is a contradictory pair within $R_1R_2R_3$. Now the use of the term *contradictory* becomes clear: the costs do not behave as could be expected from the ranks. □

The next (obvious) lemma states that contradictory chains are necessarily connected.

Lemma 3.2.12 *If there is no connection between two subchains x and y, then they cannot build a contradictory pair (x, y).*

Now we present the fact that between a contradictory pair of relations, there cannot be any other relation not connected to them without increasing cost.

Lemma 3.2.13 *Let $S = usvtw$ be a sequence. If there is no connection between relations in s and v and relations in v and t, and $rank_u(s) \geq rank_{us}(t)$, then there exists a sequence S' not having higher costs, where s immediately precedes t.*

Example 3: Consider five relations $R_1, \ldots, R_5$. The relation sizes are $|R_1| = 1$, $|R_2| = |R_3| = |R_4| = 8$, and $|R_5| = 2$. The selectivities are $f_{1,2} = \frac{1}{2}$, $f_{2,3} = \frac{1}{4}$, $f_{3,4} = \frac{1}{8}$, and $f_{4,5} = \frac{1}{2}$. Relation R_5 is not connected to relations R_2 and R_3. Further, within the sequence $R_1R_2R_5R_3R_4$ relations R_2 and R_3 have contradictory ranks: $rank_{R_1}(R_2) = \frac{4-1}{4} = \frac{3}{4}$ and $rank_{R_1R_2R_5}(R_3) = \frac{2-1}{2} = \frac{1}{2}$. Hence, at least one of $R_1R_5R_2R_3R_4$ and $R_1R_2R_3R_5R_4$ must be of no greater cost than $R_1R_2R_5R_3R_4$. This is indeed the case:

$$\begin{array}{rcccl} C(R_1R_2R_3R_5R_4) & = & 4+8+16+8 & = & 36 \\ C(R_1R_2R_5R_3R_4) & = & 4+8+16+8 & = & 36 \\ C(R_1R_5R_2R_3R_4) & = & 2+8+16+8 & = & 34 \end{array}$$

□

The next lemma shows that, if there exist two sequences of single rank-sorted relations, then their costs as well as their ranks are necessarily equal.

Lemma 3.2.14 *Let $S = x_1 \cdots x_n$ and $S' = y_1 \cdots y_n$ be two different rank-sorted chains containing exactly the relations $R_1, \ldots, R_n$, i.e.*

$$\begin{array}{rcl} rank_{x_1 \cdots x_{i-1}}(x_i) & \leq & rank_{x_1 \cdots x_i}(x_{i+1}) \text{ for all } 1 \leq i \leq n, \\ rank_{y_1 \cdots y_{i-1}}(y_i) & \leq & rank_{y_1 \cdots y_i}(y_{i+1}) \text{ for all } 1 \leq i \leq n, \end{array}$$

then S and S' have equal costs and, furthermore,

$$rank_{x_1 \cdots x_{i-1}}(x_i) = rank_{y_1 \cdots y_{i-1}}(y_i) \text{ for all } 1 < i \leq n$$

One could conjecture that the following generalization of Lemma 3.2.14 is true, although no one has proved it so far.

Conjecture 3.2.1 *Let $S = x_1 \cdots x_n$ and $S' = y_1 \cdots y_m$ be two different rank-sorted chains for the relations $R_1 \ldots, R_n$ where the $x_i's$ and $y_i's$ are subsequences such that*

$$\begin{array}{rcl} rank_{x_1 \cdots x_{i-1}}(x_i) & \leq & rank_{x_1 \cdots x_i}(x_{i+1}) \text{ for all } 1 \leq i < n, \\ rank_{y_1 \cdots y_{i-1}}(y_i) & \leq & rank_{y_1 \cdots y_i}(y_{i+1}) \text{ for all } 1 \leq i < m, \end{array}$$

and the subsequences x_i and y_j are all optimal (with respect to the fixed prefixes $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$), then S and S' have equal costs.

Consider the problem of merging two optimal unconnected chains. If we knew that the ranks of relations in an optimal chain are always sorted in ascending order, we could use the classical merge procedure to combine the two chains. The resulting chain would also be rank-sorted in ascending order and, according to Lemma 3.2.14, it would be optimal. Unfortunately, this does not work, since there are optimal chains whose ranks are not sorted in ascending order: those containing sequences with contradictory ranks.

Now, as shown in Lemma 3.2.13, between contradictory pairs of relations there cannot be any other relation not connected to them. Hence, in the merging process, we have to take care that we do not merge a contradictory pair of relations with a relation not connected to the pair. In order to achieve this, we apply the same trick as in the IKKBZ algorithm: we tie the relations of a contradictory subchain together by

building a *compound relation*. Assume that we tie together the relations $r_1, \ldots, r_n$ to a new relation $r_{1,\ldots,n}$. Then we define the size of $r_{1,\ldots,n}$ as $|r_{1,\ldots,n}| = |r_1 \bowtie \ldots \bowtie r_n|$ Further, if some r_i ($1 \leq i \leq n$) does have a connection to some $r_k \notin \{r_1, \ldots, r_n\}$ then we define the selectivity factor $f_{r_{1,\ldots,n},r_k}$ between r_k and $r_{1,\ldots,n}$ as $f_{r_{1,\ldots,n},r_k} = f_{i,k}$.

If we tie together contradictory pairs, the resulting chain of compound relations still does not have to be rank-sorted with respect to the compound relations. To overcome this, we iterate the process of tying contradictory pairs of compound relations together until the sequence of compound relations is rank-sorted, which will eventually be the case. That is, we apply the *normalization* as used in the IKKBZ algorithm. However, we have to reformulate it for relativized costs and ranks:

```
Normalize(p,s)
  while (there exist subsequences u,v (u ≠ ε) and
         compound relations x,y such that s = uxyv
         and C_pu(xy) ≤ C_pu(yx)
         and rank_pu(x) > rank_pux(y)) {
      replace xy by a compound relation (x,y);
  }
  return (p,s);
```

The compound relations in the result of the procedure `Normalize` are called *contradictory chains*. A *maximal contradictory subchain* is a contradictory subchain that cannot be made longer by further tying steps. Resolving the tyings introduced in the procedure `normalize` is called *de-normalization*. It works the same way as in the IKKBZ algorithm. The cost, size and rank functions can now be extended to sequences containing compound relations in a straightforward way. We define the cost of a sequence containing compound relations to be identical with the cost of the corresponding de-normalized sequence. The size and rank functions are defined analogously.

The following simple observation is central to the algorithms: every chain can be decomposed into a sequence of adjacent maximal contradictory subchains. For convenience, we often speak of chains instead of subchains and of contradictory chains instead of maximal contradictory subchains. The meaning should be clear from the context. Further, we note that the decomposition into adjacent maximal contradictory subchains is not unique. For example, consider an optimal subchain $r_1 r_2 r_3$ and a sequence u of preceding relations. If $rank_u(r_1) > rank_{ur_1}(r_2) > rank_{ur_1r_2}(r_3)$ one can easily show that both $(r_1, (r_2, r_3))$ and $((r_1, r_2), r_3)$ are contradictory subchains. Nevertheless, this ambiguity is not important since in the following we are only interested in contradictory subchains which are *optimal*. In this case, the condition $C_u(xy) \leq C_u(yx)$ is certainly true and can therefore be neglected. One can show that for the case of optimal subchains the indeterministically defined normalization process is well-defined, that is, if S is optimal, `normalize(P,S)` will always terminate with a unique "flat" decomposition of S into maximal contradictory subchains (flat means that we remove all but the outermost parenthesis, e.g. $(R_1R_2)(((R_5R_4)R_3)R_6)$ becomes $(R_1R_2)(R_5R_4R_3R_6)$).

The next two lemmata and the conjecture show a possible way to overcome the problem that if we consider cross products, we have an unconstrained ordering problem and the idea of Monma and Sidney as exploited in the IKKBZ algorithm is no longer applicable. The next lemma is a direct consequence of the normalization procedure.

Lemma 3.2.15 *Let $S = s_1 \dots s_m$ be an optimal chain consisting of the maximal contradictory subchains $s_1, \dots, s_m$ (as determined by the function* `normalize`*). Then*

$$rank(s_1) \leq rank_{s_1}(s_2) \leq rank_{s_1 s_2}(s_3)$$

$$\leq \dots \leq rank_{s_1 \dots s_{m-1}}(s_m),$$

in other words, the (maximal) contradictory subchains in an optimal chain are always sorted by ascending ranks.

The next result shows how to build an optimal sequence from two optimal non-interconnected sequences.

Lemma 3.2.16 *Let x and y be two optimal sequences of relations where x and y are not interconnected. Then the sequence obtained by merging the maximal contradictory subchains in x and y (as obtained by* `normalize`*) according to their ascending rank is optimal.*

Merging two sequences in the way described in Lemma 3.2.16 is a fundamental process. We henceforth refer to it by simply saying that we *merge by the ranks*.

We strongly conjecture that the following generalization of Lemma 3.2.14 is true, although it is yet unproven. It uses the notion of *optimal recursive decomposable subchains* defined in the next subsection.

Conjecture 3.2.2 *Consider two sequences S and T containing exactly the relations R_1,...,R_n. Let $S = s_1 \dots s_k$ and $T = t_1 \dots t_l$ be such that each of the maximal contradictory subchains $s_i, i = 1, \dots, k$ and $t_j, j = 1, \dots, l$ are optimal recursively decomposable. Then S and T have equal costs.*

The first algorithm

We first use a slightly modified cost function C', which additionally respects the size of the first relation in the sequence, i.e. C and C' relate via

$$C'_u(s) = \begin{cases} C(s) + |n_R|, & \text{if } u = \epsilon \text{ and } s = Rs' \\ C_u(s), & \text{otherwise} \end{cases}$$

This cost function can be treated in a more elegant way than C. The new rank function is now defined as $rank_u(s) := (T_u(s) - 1)/C'_u(s)$. Note that the rank function is now defined even if $u = \epsilon$ and s is a single relation. The size function remains unchanged. At the end of this subsection, we describe how our results can be adapted to the original cost function C.

The rank of a contradictory chain depends on the relative position of the relations that are directly connected to it. For example, the rank of the contradictory

subchain $(R_5R_3R_4R_2)$ depends on the position of the neighbouring relations R_1 and R_6 relative to $(R_5R_3R_4R_2)$. That is, whether they appear before or after the sequence $(R_5R_3R_4R_2)$. Therefore, we introduce the following fundamental definitions:

Definition 3.2.17 (neighbourhood) *We call the set of relations that are directly connected to a subchain (with respect to the query graph G) the* complete neighbourhood *of that subchain. A* neighbourhood *is a subset of the complete neighbourhood. The* complement *of a neighbourhood u of a subchain s is defined as $v \setminus u$, where v denotes the complete neighbourhood of s.*

Note that the neighbourhood of a subchain s within a larger chain us is uniquely determined by the subsequence u of relations preceding it. For convenience, we will often use sequences of preceding relations to specify neighbourhoods. We henceforth denote a pair consisting of a connected sequence s and a neighbourhood u by $[s]_u$.

Definition 3.2.18 (contradictory subchain, extent) *A* contradictory subchain $[s]_u$ *is inductively defined as follows.*

1. *For a single relation s, $[s]_u$ is a contradictory subchain.*
2. *There is a decomposition $s = vw$ such that (v, w) is a contradictory pair with respect to the preceding subsequence u and both $[v]_u$ and $[w]_{uv}$ are contradictory subchains themselves.*

The extent *of a contradictory chain $[s]_u$ is defined as the pair consisting of the neighbourhood u and the set of relations occurring in s. Since contradictory subchains are connected, the set of occurring relations has always the form $\{R_i, R_{i+1}, \ldots, R_{i+l}\}$ for some $1 \leq i \leq n,\ 0 \leq l \leq n - i$. An* optimal contradictory subchain *to a given extent is a contradictory subchain with lowest cost among all contradictory subchains of the same extent.*

The number of different extents of contradictory subchains for a chain query of n relations is $2n^2 - 2n + 1$. Each contradictory chain can be completely recursively decomposed into adjacent pairs of connected subchains. Subchains with this property are defined next (similar types of decompositions occur in [425, 768]).

Definition 3.2.19 ((optimal) recursively decomposable subchain) *A* recursively decomposable subchain $[s]_u$ *is inductively defined as follows.*

1. *If s is a single relation, then $[s]_u$ is recursively decomposable.*
2. *There is a decomposition $s = vw$ such that v is connected to w and both $[v]_u$ and $[w]_{uv}$ are recursively decomposable subchains.*

The extent of a recursively decomposable chain is defined in the same way as for contradictory chains. Note that every contradictory subchain is recursively decomposable. Consequently, the set of all contradictory subchains for a certain extent is a subset of all recursively decomposable subchains of the same extent.

Example 4: Consider the sequence of relations

$$s = R_2R_4R_3R_6R_5R_1.$$

Using parentheses to indicate the recursive decompositions, we have the following two possibilities

$$(((R_2(R_4R_3))(R_6R_5))R_1)$$

$$((R_2((R_4R_3)(R_6R_5)))R_1)$$

The extent of the recursively decomposable subchain $R_4R_3R_6R_5$ of s is $(\{R_2\}, \{R_3, R_4, R_5, R_6\})$. □

The number of different recursively decomposable chains involving the relations $R_1, \ldots, R_n$ is r_n, where r_n denotes the n-th Schröder number [768]. Hence, the total number of recursively decomposable chains is $r_n + 2(n-1)r_{n-1} + 4\sum_{i=1}^{n-2}\binom{n-2}{i}r_i$. It can be shown that

$$r_n \approx \frac{C(2+\sqrt{8})^n}{n^{3/2}}$$

where $C = 1/2\sqrt{\frac{2\sqrt{2}-4}{\pi}}$. Using Stirling's formula for $n!$ it is easy to show that $\lim_{n\to\infty}\frac{r_n}{n!} = 0$. Thus, the probability of a random permutation to be recursively decomposable strives to zero for large n.

An *optimal recursively decomposable subchain* to a given extent is a recursively decomposable subchain with lowest cost among all recursively decomposable subchains of the same extent. There is an obvious dynamic programming algorithm to compute optimal recursive decomposable subchains. It is not hard to see that *Bellman's optimality principle* [591, 206] holds and every optimal recursively decomposable subchain can be decomposed into smaller optimal recursively decomposable subchains.

Example 5: In order to compute an optimal recursively decomposable subchain for the extent

$$(\{R_2, R_7\}, \{R_3, R_4, R_5, R_6\})$$

the algorithm makes use of optimal recursively decomposable subchains for the extents

$$\begin{array}{rl}
(\{R_2\}, \{R_3\}) & (\{R_7, R_3\}, \{R_4, R_5, R_6\}) \\
(\{R_2\}, \{R_3, R_4\}) & (\{R_7, R_4\}, \{R_5, R_6\}) \\
(\{R_2\}, \{R_3, R_4, R_5\}) & (\{R_5, R_7\}, \{R_6\}) \\
(\{R_7\}, \{R_4, R_5, R_6\}) & (\{R_2, R_4\}, \{R_3\}) \\
(\{R_7\}, \{R_5, R_6) & (\{R_2, R_5\}, \{R_3, R_4\}) \\
(\{R_7\}, \{R_6\}) & (\{R_2, R_6\}, \{R_3, R_4, R_5\})
\end{array}$$

which have been computed in earlier steps[2]. A similar dynamic programming algorithm can be used to determine optimal contradictory subchains. □

Let E be the set of all possible extents. We define the following partial order $\mathcal{P} = (E, \prec)$ on E. For all extents $e_1, e_2 \in E$, we have $e_1 \prec e_2$ if and only if

[2]The splitting of extents induces a partial order on the set of extents.

e_1 can be obtained by splitting the extent e_2. For example, $(\{R_7\}, \{R_5, R_6\}) \prec (\{R_2, R_7\}, \{R_3, R_4, R_5, R_6\})$. The set of maximal extents M then corresponds to a set of incomparable elements (antichain) in $\mathcal{P}$ such that for all extents e enumerated so far, there is an extent $e' \in M$ with $e \prec e'$.

Now, since every optimal join sequence has a representation as a sequence of contradictory subchains, we only have to determine this representation. Consider a contradictory subchain c in an optimal join sequence s. What can we say about c? Obviously, c has to be optimal with respect to the neighbourhood defined by the relations preceding c in s. Unfortunately, identifying contradictory subchains that are optimal sequences seems to be as hard as the whole problem of optimizing chain queries. Therefore, we content ourselves with the following weaker condition which may lead to multiple representations. Nevertheless, it seems to be the strongest condition for which all subchains satisfying the condition can be computed in polynomial time. The condition says that s should be optimal both with respect to all contradictory chains of the same extent as s and with respect to all recursively decomposable subchains of the same extent. So far it is not clear whether these conditions lead to multiple representations. Therefore, we have no choice but to enumerate all possible representations and select the one with minimal costs. Next we describe the first algorithm.

Algorithm Chain-I':

1. Use dynamic programming to determine all optimal contradictory subchains. This step can be made faster by keeping track of the set M of all maximal extents (with respect to the partial order induced by splitting extents).
2. Determine all optimal recursively decomposable subchains for all extents included in some maximal extent in M.
3. Compare the results from steps 1 and 2 and retain only matching subchains.
4. Sort the contradictory subchains according to their ranks.
5. Eliminate contradictory subchains that cannot be part of a solution.
6. Use backtracking to enumerate all sequences of rank-ordered optimal contradictory subchains and keep track of the sequence with lowest cost.

In step 5 of the algorithm, we eliminate contradictory subchains that do not contribute to a solution. Note that the contradictory subchains in an optimal sequence are characterized by the following two conditions.

1. The extents of all contradictory subchains in the representation build a partition of the set of all relations.

2. The neighbourhoods of all contradictory subchains are consistent with the relations occurring at earlier and later positions in the sequence.

Note that any contradictory subchain occurring in the optimal sequence (except at the first and last positions) necessarily has matching contradictory subchains preceding and succeeding it in the list. In fact, every contradictory subchain X occurring in the optimal join sequence must satisfy the following two conditions.

1. For every relation R in the neighbourhood of X, there exists a contradictory subchain Y at an earlier position in the list which itself meets condition 1, such that R occurs in Y, and Y can be followed by X.

2. For every relation R in the complementary neighbourhood of X, there exists a contradictory subchain Y at a later position in the list which itself meets condition 2, such that R occurs in the neighbourhood of Y, and X can be followed by Y.

Using these two conditions, we can eliminate "useless" contradictory chains from the rank-ordered list by performing a reachability algorithm for each of the DAGs defined by the conditions 1 and 2. In the last step of our algorithm, backtracking is used to enumerate all representations. Suppose that at some step of the algorithm we have determined an initial sequence of contradictory subchains and have a rank-sorted list of the remaining possible contradictory subchains. In addition to the two conditions mentioned above, another reachability algorithm can be applied to determine the set of reachable relations from the list (with respect to the given prefix). With the use of this information, all branches that do not lead to a complete join sequence can be pruned.

Let us analyze the worst case time complexity of the algorithm. The two dynamic programming steps both iterate over $O(n^2)$ different extents, and each extent gives rise to $O(n)$ splittings. Moreover, for each extent one normalization is necessary, which requires linear time (cost, size and rank can be computed in constant time using recurrences). Therefore, the complexity of the two dynamic programming steps is $O(n^4)$. Sorting $O(n^2)$ contradictory chains can be done in time $O(n^2 \log n)$. The step where all "useless" contradictory subchains are eliminated, consists of two stages of a reachability algorithm which has complexity $O(n^4)$. If conjecture 3.2.2 is true, the backtracking step requires linear time, and the total complexity of the algorithm is $O(n^4)$. Otherwise, if conjecture 3.2.2 is false, the algorithm might exhibit exponential worst case time complexity.

We now describe how to reduce the problem for our original cost function C to the problem for the modified cost function C'. One difficulty with the original cost function is that the ranks are defined only for subsequences of *at least two* relations. Hence, for determining the first relation in our solution we do not have sufficient information. An obvious solution to this problem is to try every relation as starting relation, process each of the two resulting chain queries separately and choose the chain with minimum costs. The new complexity will increase by about a factor of n. This first approach is not very efficient, since the dynamic programming computations overlap considerably, e.g. if we perform dynamic programming on the two overlapping chains $R_1R_2R_3R_4R_5R_6$ and $R_2R_3R_4R_5R_6R_7$, for the intersecting chain $R_2R_3R_4R_5R_6$ everything is computed twice. The cue is that we can perform the dynamic programming calculations before we consider a particular starting relation. Hence, the final algorithm can be sketched as follows:

Algorithm CHAIN-I:

1. Compute all optimal contradictory chains by dynamic programming (corresponds to the steps 1-4 of Algorithm I')

2. For each starting relation R_i, perform the following steps:

   (a) Let L_1 be the result of applying steps 5 and 6 of Algorithm I' to all contradictory subchains whose extent (N, M) satisfies $R_i \in N$ and $M \subseteq \{R_1, \ldots, R_i\}$.

(b) Let L_2 be the result of applying steps 5 and 6 of Algorithm I' to all contradictory subchains whose extent (N, M) satisfies $R_i \in N$ and $M \subseteq \{R_i, \ldots, R_n\}$.

(c) For all $(l_1, l_2) \in L_1 \times L_2$, perform the following steps:

i. Let L be the result of merging l_1 and l_2 according to their ranks.

ii. Use $R_i L$ to update the current-best join ordering.

Suppose that conjecture 3.2.2 is true, and we can replace the backtracking part by a search for the first solution. Then the complexity of the step 1 is $O(n^4)$, whereas the complexity of step 2 amounts to $\sum_{i=1}^{n}(O(i^2) + O(n-i)^2 + O(n)) = O(n^3)$. Hence, the total complexity would be $O(n^4)$ in the worst case. Of course, if our conjecture is false, the necessary backtracking step might lead to an exponential worst case complexity.

The second algorithm

The second algorithm is much simpler than the first one but proves to be less efficient in practice. Since the new algorithm is very similar to some parts of the old one, we just point out the differences between both algorithms. The new version of the algorithm works as follows.

Algorithm CHAIN-II':

1. Use dynamic programming to compute an optimal recursive decomposable chain for the whole set of relations $\{R_1, \ldots, R_n\}$.

2. Normalize the resulting chain.

3. Reorder the contradictory subchains according to their ranks.

4. De-normalize the sequence.

Step 1 is identical to step 2 of our first algorithm. Note that Lemma 3.2.15 cannot be applied to the sequence in Step 2, since an optimal recursive decomposable chain is not necessarily an optimal chain. Therefore, the question arises whether Step 3 really makes sense. One can show that the partial order defined by the precedence relation among the contradictory subchains has the property that all elements along paths in the partial order are sorted by rank. By computing a greedy topological ordering (greedy with respect to the ranks), we obtain a sequence as requested in step 3.

Let us briefly analyze the worst case time complexity of the second algorithm. The first step requires time $O(n^4)$, whereas the second step requires time $O(n^2)$. The third step has complexity $O(n \log n)$. Hence, the total complexity is $O(n^4)$.

Algorithm II' is based on the cost function C'. We can now modify the algorithm for the original cost function C as follows.

Algorithm CHAIN-II:

1. Compute all optimal recursive decomposable chains by dynamic programming (corresponds to step 1 of Algorithm II')

2. For each starting relation R_i, perform the following steps:

(a) Let L_1 be the result of applying the steps 2 and 3 of Algorithm II' to all optimal recursive decomposable subchains whose extent (N, M) satisfies $R_i \in N$ and $M \subseteq \{R_1, \ldots, R_i\}$.

(b) Let L_2 be the result of applying the steps 2 and 3 of Algorithm II' to all optimal recursive decomposable subchains whose extent (N, M) satisfies $R_i\ N$ and $M \subseteq \{R_i, \ldots, R_n\}$.

(c) Let L be the result of merging L_1 and L_2 according to their ranks.

(d) De-normalize L.

(e) Use $R_i L$ to update the current-best join ordering.

The complexity of Step 1 is $O(n^4)$, whereas the complexity of Step 2 amounts to $\sum_{i=1}^{n}(O(i^2) + O(n-i)^2 + O(n)) = O(n^3)$. Hence, the time complexity of Algorithm II is $O(n^4)$.

Summarizing, we are now left with one algorithm that produces the optimal result but whose worst-case runtime behavior is unknown and one algorithm with polynomial runtime but producing a result which has not been proven to be optimal. Due to this lack of hard facts, Moerkotte and Scheufele ran about 700,000 experiments with random queries of sizes up to 30 relations and fewer experiments for random queries with up to 300 relations to compare the results of our algorithms. For $n \leq 15$, they additionally compared the results with a standard dynamic programming algorithm. Their findings can be summarized as follows.

- All algorithms yielded identical results.
- Backtracking always led to exactly one sequence of contradictory chains.
- In the overwhelming majority of cases the first algorithm proved to be faster than the second.

Whereas the run time of the second algorithm is mainly determined by the number of relations in the query, the run time of the first also heavily depends on the number of existing optimal contradictory subchains. In the worst case, the first algorithm is slightly inferior to the second. Additionally, Hamalainen reports on an independent implementation of the second algorithm [388]. He could not find an example where the second algorithm did not produce the optimal result either. We encourage the reader to prove that it produces the optimal result.

EX

3.2.8 Transformation-Based Approaches

The idea of transformation-based algorithms can be described as follows. Starting from an arbitrary join tree, equivalences (such as commutativity and associativity) are applied to it to derive a set of new join trees. For each of the join trees, the equivalences are again applied to derive even more join trees. This procedure is repeated until no new join tree can be derived. This procedure exhaustively enumerates the set of all bushy trees. Furthermore, before an equivalence is applied, it is difficult to see whether the resulting join tree has already been produced or not (see also Figure 2.6). Thus, this procedure is highly inefficient. Hence, it does not play any role in practice. Nevertheless, we give the pseudo-code for it, since it forms the basis for several of the following

algorithms. We split the exhaustive transformation approach into two algorithms. One that applies all equivalences to a given join tree (`ApplyTransformations`) and another that does the loop (`ExhaustiveTransformation`). A transformation is applied in a directed way. Thus, we reformulate commutativity and associativity as rewrite rules using $\rightsquigarrow$ to indicate the direction.

The following table summarizes all rules commonly used in transformation-based and randomized join ordering algorithms. The first three are directly derived from the commutativity and associativity laws for the join. The other rules are shortcuts used under special circumstances. For example, left associativity may turn a left-deep tree into a bushy tree. When only left-deep trees are to be considered, we need a replacement for left associativity. This replacement is called left join exchange.

$R_1 \bowtie R_2$	$\rightsquigarrow$	$R_2 \bowtie R_1$	Commutativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$R_1 \bowtie (R_2 \bowtie R_3)$	Right Associativity
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$(R_1 \bowtie R_2) \bowtie R_3$	Left Associativity
$(R_1 \bowtie R_2) \bowtie R_3$	$\rightsquigarrow$	$(R_1 \bowtie R_3) \bowtie R_2$	Left Join Exchange
$R_1 \bowtie (R_2 \bowtie R_3)$	$\rightsquigarrow$	$R_2 \bowtie (R_1 \bowtie R_3)$	Right Join Exchange

Two more rules are often used to transform left-deep trees. The first operation (*swap*) exchanges two arbitrary relations in a left-deep tree. The second operation (*3Cycle*) performs a cyclic rotation of three arbitrary relations in a left-deep tree. To account for different join methods, a rule called *join method exchange* is introduced.

The first rule set (*RS-0*) we are using contains the commutativity rule and both associativity rules. Applying associativity can lead to cross products. If we do not want to consider cross products, we only apply any of the two associativity rules if the resulting expression does not contain a cross product. It is easy to extend `ApplyTransformations` to cover this by extending the `if` conditions with RS-0

```
and (ConsiderCrossProducts || connected(·))
```

where the argument of `connected` is the result of applying a transformation.

```
ExhaustiveTransformation({R_1, ..., R_n})
Input:  a set of relations
Output:  an optimal join tree
Let T be an arbitrary join tree for all relations
Done = ∅; // contains all trees processed
ToDo = {T}; // contains all trees to be processed
while (!empty(ToDo)) {
  Let T be an arbitrary tree in ToDo
  ToDo \= T;
  Done ∪= T;
  Trees = ApplyTransformations(T);
  for all T ∈ Trees do {
    if (T ∉ ToDo ∪ Done) {
      ToDo += T;
    }
```

```
  }
}
return cheapest tree found in Done;

ApplyTransformations(T)
Input:  join tree
Output:  all trees derivable by associativity and commutativity
Trees = ∅;
Subtrees = all subtrees of T rooted at inner nodes
for all S ∈ Subtrees do {
  if (S is of the form S1 ⋈ S2) {
    Trees += S2 ⋈ S1;
  }
  if (S is of the form (S1 ⋈ S2) ⋈ S3) {
    Trees += S1 ⋈ (S2 ⋈ S3);
  }
  if (S is of the form S1 ⋈ (S2 ⋈ S3)) {
    Trees += (S1 ⋈ S2) ⋈ S3;
  }
}
return Trees;
```

Besides the problems mentioned above, this algorithm also has the problem that the sharing of subtrees is a non-trivial task. In fact, we assume that `ApplyTransformations` produces modified copies of T. To see how `ExhaustiveTransformation` works, consider again Figure 2.6. Assume that the top-left join tree is the initial join tree. Then, from this join tree `ApplyTransformations` produces all trees reachable by some edge. All of these are then added to `ToDo`. The next call to `ApplyTransformations` with any to the produced join trees will have the initial join tree contained in `Trees`. The complete set of visited join trees after this step is determined from the initial join tree by following at most two edges.

Let us reformulate the algorithm such that it uses a data structure similar to dynamic programming or memoization in order to avoid duplicate work. For any subset of relations, dynamic programming remembers the best join tree. This does not quite suffice for the transformation-based approach. Instead, we have to keep all join trees generated so far including those differing in the order of the arguments or a join operator. However, subtrees can be shared. This is done by keeping pointers into the data structure (see below). So, the difference between dynamic programming and the transformation-based approach becomes smaller. The main remaining difference is that dynamic programming only considers these join trees while with the transformation-based approach we have to keep the considered join trees since other join trees (more beneficial) might be generatable from them.

The data structure used for remembering trees is often called the MEMO structure. For every subset of relations to be joined (except the empty set), a *class* exists in the MEMO structure. Each class contains all the join trees that join exactly the relations describing the class. Here is an example for join trees containing three relations.

$\{R_1, R_2, R_3\}$	$\{R_1, R_2\} \bowtie R_3, R_3 \bowtie \{R_1, R_2\}$, $\{R_1, R_3\} \bowtie R_2, R_2 \bowtie \{R_1, R_3\}$, $\{R_2, R_3\} \bowtie R_1, R_1 \bowtie \{R_2, R_3\}$
$\{R_2, R_3\}$	$\{R_2\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_2\}$
$\{R_1, R_3\}$	$\{R_1\} \bowtie \{R_3\}, \{R_3\} \bowtie \{R_1\}$
$\{R_1, R_2\}$	$\{R_1\} \bowtie \{R_2\}, \{R_2\} \bowtie \{R_1\}$
$\{R_3\}$	R_3
$\{R_2\}$	R_2
$\{R_1\}$	R_1

Here, we used the set notation $\{\ldots\}$ as an argument to a join to denote a reference to the class of join trees joining the relations contained in it.

We reformulate our transformation-based algorithm such that it fills in and uses the MEMO structure [655]. In a first step, the MEMO structure is initialized by creating an arbitrary join tree for the class $\{R_1, \ldots, R_n\}$ and then going down this join tree and creating an entry for every join encountered. Then, we call `ExploreClass` on the root class comprising all relations to be joined. `ExploreClass` then applies `ApplyTransformations2` to every member of the class it is called upon. `ApplyTransformations2` then applies all rules to generate alternatives.

```
ExhaustiveTransformation2(Query Graph G)
Input: a query specification for relations {R_1,...,R_n}.
Output: an optimal join tree
  initialize MEMO structure
  ExploreClass({R_1,...,R_n})
  return best of class {R_1,...,R_n}

ExploreClass(C)
Input: a class C ⊆ {R_1,...,R_n}
Output: none, but has side-effect on MEMO-structure
  while (not all join trees in C have been explored) {
    choose an unexplored join tree T in C
    ApplyTransformation2(T)
    mark T as explored
  }
  return

ApplyTransformations2(T)
Input: a join tree of a class C
Output: none, but has side-effect on MEMO-structure
  ExploreClass(left-child(T));
  ExploreClass(right-child(T));
  foreach transformation T and class member of child classes {
    foreach T' resulting from applying T to T {
```

```
      if T' not in MEMO structure {
        add T' to class C of MEMO structure
      }
    }
  }
  return
```

`ApplyTransformations2` uses a set of transformations to be applied. We discuss now the effect of different transformation sets on the complexity of the algorithm. Applying `ExhaustiveTransformation2` with a rule set consisting of Commutativity and Left and Right Associativity generates $4^n - 3^{n+1} + 2^{n+2} - n - 2$ duplicates for n relations. Contrast this with the number of join trees contained in a completely filled MEMO structure[3]: $3^n - 2^{n+1} + n + 1$. This clearly shows the problem.

The problem of generating the same join tree several times was considered by Pellenkoft, Galindo-Legaria, and Kersten [655, 656, 657]. The solution lies in parameterizing `ExhaustiveTransformation2` by an appropriate set of transformations. The basic idea is to remember for every join operator which rules are applicable to it. For example, after applying commutativity to a join operator, we disable commutativity for it.

For acyclic queries, the following rule set guarantees that all bushy join trees are generated, but no duplicates [657]. Thereby, cross products are not considered. That is, a rule is only applicable if it does not result in a cross product. This restricts the applicability of the above algorithm to connected queries. We use C_i to denote some
RS-1 class of the MEMO structure. We call the following rule set RS-1:

T_1**: Commutativity** $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, and T_3 for $\bowtie_1$.

T_2**: Right Associativity** $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2 and T_3 for $\bowtie_2$ and enable all rules for $\bowtie_3$.

T_3**: Left associativity** $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2 and T_3 for $\bowtie_3$ and enable all rules for $\bowtie_2$.

In order to be able to follow these rules, the procedure `ApplyTransformations2` has to be enhanced such that it is able to keep track of the application history of the rules for every join operator. The additional memory requirement is neglectible, since a single bit for each rules suffices.

As an example, let us consider the chain query $R_1 - R_2 - R_3 - R_4$. Figure 3.14 shows the MEMO structure. The first column gives the sets of the relations identifying each class. We leave out the single relation classes assuming that $\{R_i\}$ has R_i as its only join tree which is marked as explored.

The second column shows the initialization with an arbitrarily chosen join tree. The third column is the one filled by the `Apply Transformation2` procedure. We apply the rule set RS-1, which consists of three transformations. Each join is annotated with three bits, where the i-th bit indicates whether T_i is applicable (1) or not

[3]The difference to the according number for dynamic programming is due to the fact that we have to keep alternatives generated by commutativity and that join trees for single relations are counted.

Class	Initialization	Transformation	Step
$\{R_1, R_2, R_3, R_4\}$	$\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$	$\{R_3, R_4\} \bowtie_{000} \{R_1, R_2\}$	3
		$R_1 \bowtie_{100} \{R_2, R_3, R_4\}$	4
		$\{R_1, R_2, R_3\} \bowtie_{100} R_4$	5
		$\{R_2, R_3, R_4\} \bowtie_{000} R_1$	8
		$R_4 \bowtie_{000} \{R_1, R_2, R_3\}$	10
$\{R_2, R_3, R_4\}$		$R_2 \bowtie_{111} \{R_3, R_4\}$	4
		$\{R_3, R_4\} \bowtie_{000} R_2$	6
		$\{R_2, R_3\} \bowtie_{100} R_4$	6
		$R_4 \bowtie_{000} \{R_2, R_3\}$	7
$\{R_1, R_3, R_4\}$			
$\{R_1, R_2, R_4\}$			
$\{R_1, R_2, R_3\}$		$\{R_1, R_2\} \bowtie_{111} R_3$	5
		$R_3 \bowtie_{000} \{R_1, R_2\}$	9
		$R_1 \bowtie_{100} \{R_2, R_3\}$	9
		$\{R_2, R_3\} \bowtie_{000} R_1$	9
$\{R_3, R_4\}$	$R_3 \bowtie_{111} R_4$	$R_4 \bowtie_{000} R_3$	2
$\{R_2, R_4\}$			
$\{R_2, R_3\}$			
$\{R_1, R_4\}$			
$\{R_1, R_3\}$			
$\{R_1, R_2\}$	$R_1 \bowtie_{111} R_2$	$R_2 \bowtie_{000} R_1$	1

Figure 3.14: Example of rule transformations (RS-1)

(0). After initializing the MEMO structure, `ExhaustiveTransformation2` calls `ExploreClass` for $\{R_1, R_2, R_3, R_4\}$. The only (unexplored) join tree is $\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$, which will become the argument of `ApplyTransformations2`. Next, `ExploreClass` is called on $\{R_1, R_2\}$ and $\{R_3, R_4\}$. In both cases, T_1 is the only applicable rule, and the result is shown in the third column under steps 1 and 2. Now we have to apply all transformations on $\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$. Commutativity T_1 gives us $\{R_3, R_4\} \bowtie_{000} \{R_1, R_2\}$ (Step 3). For right associativity, we have two elements in class $\{R_1, R_2\}$. Substituting them and applying T_2 gives

1. $(R_1 \bowtie R_2) \bowtie \{R_3, R_4\} \rightsquigarrow R_1 \bowtie_{100} (R_2 \bowtie_{111} \{R_3, R_4\})$
2. $(R_2 \bowtie R_1) \bowtie \{R_3, R_4\} \rightsquigarrow R_2 \bowtie_{111} (R_1 \times \{R_3, R_4\})$

The latter contains a cross product. This leaves us with the former as the result of Step 4. The right argument of the top most join is $R_2 \bowtie_{111} \{R_3, R_4\}$. Since we do not find it in class $\{R_2, R_3, R_4\}$, we add it (4).

T_3 is next.

1. $\{R_1, R_2\} \bowtie (R_3 \bowtie R_4) \rightsquigarrow (\{R_1, R_2\} \bowtie_{111} R_3) \bowtie_{100} R_4$
2. $\{R_1, R_2\} \bowtie (R_4 \bowtie R_3) \rightsquigarrow (\{R_1, R_2\} \times R_4) \bowtie_{100} R_3$

The latter contains a cross product. This leaves us with the former as the result of Step 5. We also add $\{R_1, R_2\} \bowtie_{111} R_3$. Now that $\{R_1, R_2\} \bowtie_{111} \{R_3, R_4\}$ is completely explored, we turn to $\{R_3, R_4\} \bowtie_{000} \{R_1, R_2\}$, but all transformations are disabled here.

$R_1 \bowtie_{100} \{R_2, R_3, R_4\}$ is next. First, $\{R_2, R_3, R_4\}$ has to be explored. The only entry is $R_2 \bowtie_{111} \{R_3, R_4\}$. Remember that $\{R_3, R_4\}$ is already explored. T_2 is not applicable. The other two transformations give us

T_1 $\{R_3, R_4\} \bowtie_{000} R_2$

T_3 $(R_2 \bowtie_{000} R_3) \bowtie_{100} R_4$ and $(R_2 \times R_4) \bowtie_{100} R_3$

Those join trees not exhibiting a cross product are added to the MEMO structure under 6. Applying commutativity to $\{R_2, R_4\} \bowtie_{100} R_3$ gives 7. Commutativity is the only rule enabled for $R_1 \bowtie_{100} \{R_2, R_3, R_4\}$. Its application results in 8.

$\{R_1, R_2, R_3\} \bowtie_{100} R_4$ is next. It is simple to explore the class $\{R_1, R_2, R_3\}$ with its only entry $\{R_1, R_2\} \bowtie_{111} R_3$:

T_1 $R_3 \bowtie_{000} \{R_1, R_2\}$

T_2 $R_1 \bowtie_{100} (R_2 \bowtie_{111} R_3)$ and $R_2 \bowtie_{100} (R_1 \times R_3)$

Commutativity can still be applied to $R_1 \bowtie_{100} (R_2 \bowtie_{111} R_3)$. All the new entries are numbered 9. Commutativity is the only rule enabled for $\{R_1, R_2, R_3\} \bowtie_{100} R_4$ Its application results in 10.

□

The next two sets of transformations were originally intended for generating all bushy/left-deep trees for a clique query [656]. They can, however, also be used to generate all bushy trees when cross products are considered. The rule set RS-2 for bushy trees is

T_1**: Commutativity** $C_1 \bowtie_0 C_2 \rightsquigarrow C_2 \bowtie_1 C_1$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_1$.

T_2**: Right Associativity** $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow C_1 \bowtie_2 (C_2 \bowtie_3 C_3)$
Disable transformations T_2, T_3, and T_4 for $\bowtie_2$.

T_3**: Left Associativity** $C_1 \bowtie_0 (C_2 \bowtie_1 C_3) \rightsquigarrow (C_1 \bowtie_2 C_2) \bowtie_3 C_3$
Disable transformations T_2, T_3 and T_4 for $\bowtie_3$.

T_4**: Exchange** $(C_1 \bowtie_0 C_2) \bowtie_1 (C_3 \bowtie_2 C_4) \rightsquigarrow (C_1 \bowtie_3 C_3) \bowtie_4 (C_2 \bowtie_5 C_4)$
Disable all transformations T_1, T_2, T_3, and T_4 for $\bowtie_4$.

If we initialize the MEMO structure with left-deep trees, we can strip down the above rule set to Commutativity and Left Associativity. The reason is an observation made by Shapiro et al.: from a left-deep join tree we can generate all bushy trees with only these two rules [767].

If we want to consider only left-deep trees, the following rule set RS-3 is appropriate:

T_1 **Commutativity** $R_1 \bowtie_0 R_2 \rightsquigarrow R_2 \bowtie_1 R_1$
Here, the R_i are restricted to classes with exactly one relation. T_1 is disabled for $\bowtie_1$.

T_2 **Right Join Exchange** $(C_1 \bowtie_0 C_2) \bowtie_1 C_3 \rightsquigarrow (C_1 \bowtie_2 C_3) \bowtie_3 C_2$
Disable T_2 for $\bowtie_3$.

3.3 Probabilistic Algorithms

3.3.1 Generating Random Left-Deep Join Trees with Cross Products

The basic idea of the algorithms in this section and the following sections is to generate a set of randomly chosen join trees, evaluate their costs, and return the best one. The problem with this approach lies in the random generation of join trees: every join tree has to be generated with equal probability. Although there are some advocates of the pure random approach [287, 288, 290, 286], typically a random join tree or a set of random join trees is used in subsequent algorithms like iterative improvement and simulated annealing.

Obviously, if we do not consider cross products the problem is really hard, since the query graph plays an important role. So let us start with the simplest case where random join trees are generated that might contain cross products even for connected query graphs. Then, any join tree is a valid join tree.

The general idea behind all algorithms is the following. Assume that the number of join trees in the considered search space is known to be N. Then, instead of generating a random join tree directly, a bijective mapping from the interval of non-negative integers $[0, N[$ to a join tree in the search space is established. Then, a random join tree can be generated by (1) generating a random number in $[0, N[$ and (2) mapping the number to the join tree. The problem of bijectively mapping an interval of non-negative integers to elements of a set is usually called *unranking*. The opposite mapping is called *ranking*. Obviously, the crux in our case is the efficiency of the unranking problem.

We start with generating random left-deep join trees for n relations. This problem is identical to generating random permutations. That is, we look for a fast unranking algorithm that maps the non-negative integers in $[0, n![$ to permutations. Let us consider permutations of the numbers $\{0, \ldots, n-1\}$. A mapping between these numbers and relations is established easily, e.g. via an array. The traditional approach to ranking/unranking of permutations is to first define an ordering on the permutations and then find a ranking and unranking algorithm relative to that ordering. For the lexicographic order, algorithms require $O(n^2)$ time [541, 694]. More sophisticated algorithms separate the ranking/unranking algorithms into two phases. For ranking, first the *inversion vector* of the permutation is established. Then, ranking takes place for the inversion vector. Unranking works in the opposite direction. The *inversion vector* of a permutation $\pi = \pi_0, \ldots, \pi_{n-1}$ is defined to be the sequence $v = v_0, \ldots, v_{n-1}$, where v_i is equal to the number of entries π_j with $\pi_j > \pi_i$ and $j < i$. Inversion vectors uniquely determine a permutation [841]. However, naive algorithms of this approach again require $O(n^2)$ time. Better algorithms require $O(n \log n)$. Using an elaborated data structure, Dietz' algorithm requires $O((n \log n)/(\log \log n))$ [235]. Other orders like the Steinhaus-Johnson-Trotter order have been exploited for ranking/unranking

but do not yield any run-time advantage over the above mentioned algorithms (see [502, 694]).

Since it is not important for our problem that any order constraints are satisfied for the ranking/unranking functions, we use the fastest possible algorithm established by Myrvold and Ruskey [615]. It runs in $O(n)$ which is also easily seen to be a lower bound.

The algorithm is based on the standard algorithm to generate random permutations [219, 241, 607]. An array π is initialized such that $\pi[i] = i$ for $0 \leq i \leq n-1$. Then, the loop

for $(k = n-1;\ k >= 0;\ --k)$ `swap(`$\pi[k], \pi[$`random(`k`)]);`

is executed where `swap` exchanges two elements and `random(k)` generates a random number in $[0, k]$. This algorithm randomly picks any of the possible permutations. Assume the random elements produced by the algorithm are $r_{n-1}, \ldots, r_0$ where $0 \leq r_i \leq i$. Obviously, there are exactly $n(n-1)(n-2)\ldots 1 = n!$ such sequences and there is a one-to-one correspondence between these sequences and the set of all permutations. We can thus unrank $r \in [0, n![$ by turning it into a unique sequence of values $r_{n-1}, \ldots, r_0$. Note that after executing the swap with r_{n-1}, every value in $[0, n[$ is possible at position $\pi[n-1]$. Further, $\pi[n-1]$ is never touched again. Hence, we can unrank r as follows. We first set $r_{n-1} = r \mod n$ and perform the swap. Then, we define $r' = \lfloor r/n \rfloor$ and iteratively unrank r' to construct a permutation of $n-1$ elements. The following algorithm realizes this idea.

```
Unrank(n, r) {
Input:  the number n of elements to be permuted
        and the rank r of the permutation to be constructed
Output: a permutation π
  for (i = 0; i < n; ++i) π[i] = i;
  Unrank-Sub(n, r, π);
  return π;
  }
}

  Unrank-Sub(n, r, π) {
  for (i = n; i > 0; --i) {
    swap(π[i-1], π[r mod i]);
    r = ⌊r/i⌋;
  }
}
```

3.3.2 Generating Random Join Trees with Cross Products

Next, we want to randomly construct bushy plans possibly containing cross products. This is done in several steps:

1. Generate a random number b in $[0, C(n-1)[$.
2. Unrank b to obtain a bushy tree with $n-1$ inner nodes.
3. Generate a random number p in $[0, n![$.
4. Unrank p to obtain a permutation.
5. Attach the relations in order p from left to right as leaf nodes to the binary tree obtained in Step 2.

The only step that we still have to discuss is Step 2. It is a little involved and we can only try to bring across the general idea. For details, the reader is referred to the literature [541, 542, 543].

Consider Figure 3.15. It contains all 14 possible trees with four inner nodes. The trees are ordered according to the rank we will consider. The bottom-most number below any tree is its rank in $[0, 14[$. While unranking, we do not generate the trees directly, but an encoding of the tree instead. This encoding works as follows. Any binary tree corresponds to a word in a Dyck language with one pair of parenthesis. The alphabet hence consists of $\Sigma = \{'(', ')'\}$. For join trees with n inner nodes, we use Dyck words of length $2n$ whose parenthesization is correct. That is, for every $'('$, we have a subsequent $')'$. From a given join tree, we obtain the Dyck word by a preorder traversal. Whenever we encounter an inner node, we encode this with a $'('$. All but the last leaf nodes are encoded by a $')'$. Appending all these $2n$ encodings gives us a Dyck word of length $2n$. Figure 3.15 shows directly below each tree its corresponding Dyck word. In the line below, we simply changed the representation by substituting every $'('$ by a $'1'$ and every $')'$ by a $'0'$. The encoding that will be generated by the unranking algorithm is shown in the third line below each tree: we remember the places (index in the bit-string) where we find a $'1'$.

In order to do the unranking, we need to do some counting. Therefor, we map Dyck words to paths in a triangular grid. For $n = 4$ this grid is shown in Figure 3.16. We always start at $(0, 0)$ which means that we have not opened a parenthesis. When we are at (i, j), opening a parenthesis corresponds to going to $(i+1, j+1)$ and closing a parenthesis to going to $(i + 1, j - 1)$. We have thus established a bijective mapping between Dyck words and paths in the grid. Thus counting Dyck words corresponds to counting paths.

The number of different paths from $(0, 0)$ to (i, j) can be computed by

$$p(i,j) = \frac{j+1}{i+1}\binom{i+1}{\frac{1}{2}(i+j)+1}$$

These numbers are called the *Ballot numbers* [117]. The number of paths from (i, j) to $(2n, 0)$ can thus be computed as (see [542, 543]):

$$q(i,j) = p(2n-i, j)$$

Note the special case $q(0, 0) = p(2n, 0) = C(n)$. In Figure 3.16, we annotated nodes (i, j) by $p(i, j)$. These numbers can be used to assign (sub-) intervals to paths (Dyck words, trees). For example, if we are at $(4, 4)$, there exists only a single path to $(2n, 0)$. Hence, the path that travels the edge $(4, 4) \to (5, 3)$ has rank 0. From $(3, 3)$ there are

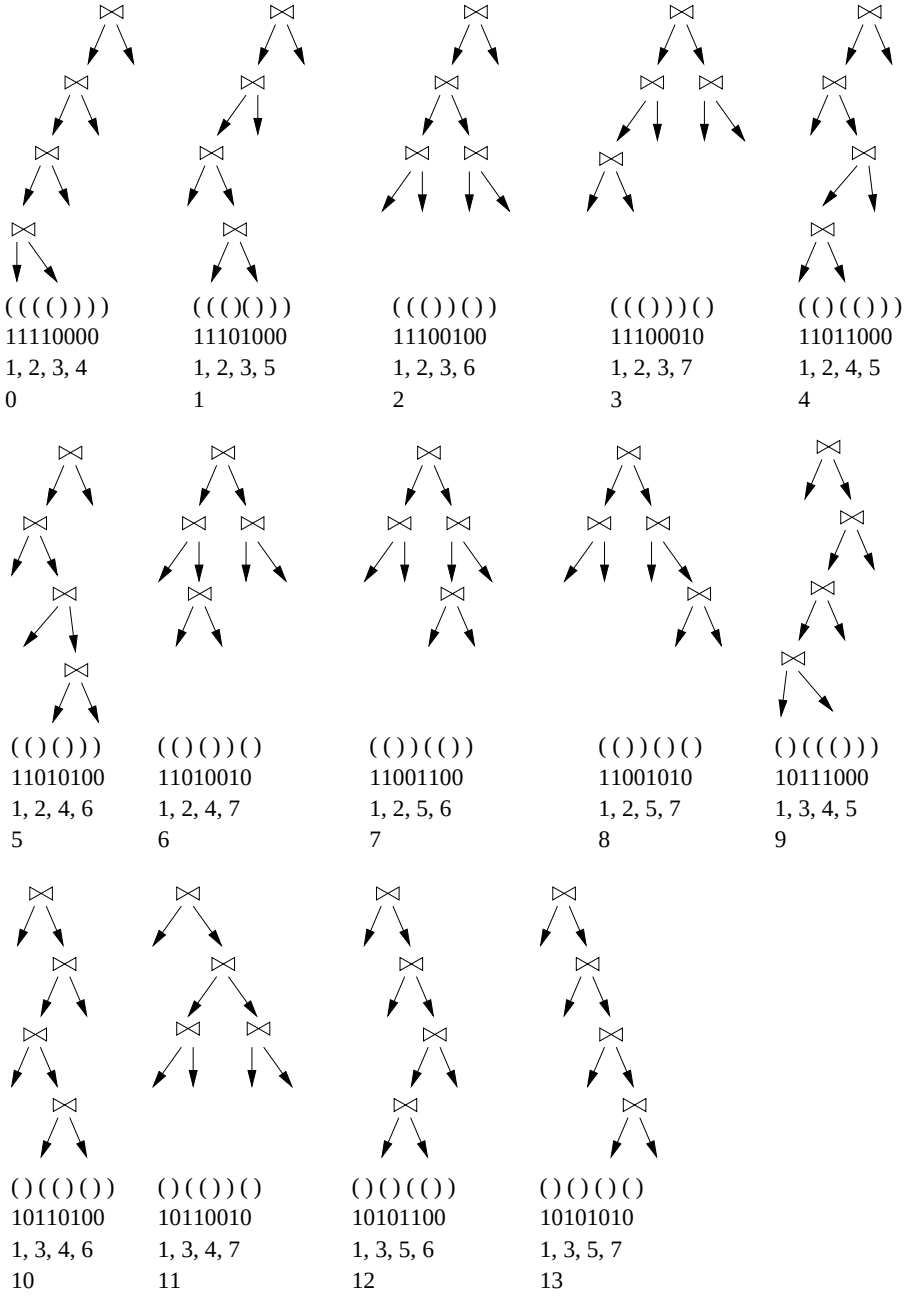


Figure 3.15: Encoding Trees

four paths to $(2n, 0)$, one of which we already considered. This leaves us with three paths that travel the edge $(3, 3) \rightarrow (4, 2)$. The paths in this part as assigned ranks in the interval $[1, 4[$. Figure 3.16 shows the intervals near the edges. For unranking, we can now proceed as follows. Assume we have a rank r. We consider opening a parenthesis (go from (i, j) to $(i + 1, j + 1)$) as long as the number of paths from that point does no longer exceed our rank r. If it does, we close a parenthesis instead (go from (i, j) to $(i - 1, j + 1)$). Assume, that we went upwards to (i, j) and then had to go down to $(i - 1, j + 1)$. We subtract the number of paths from $(i + 1, j + 1)$ from our rank r

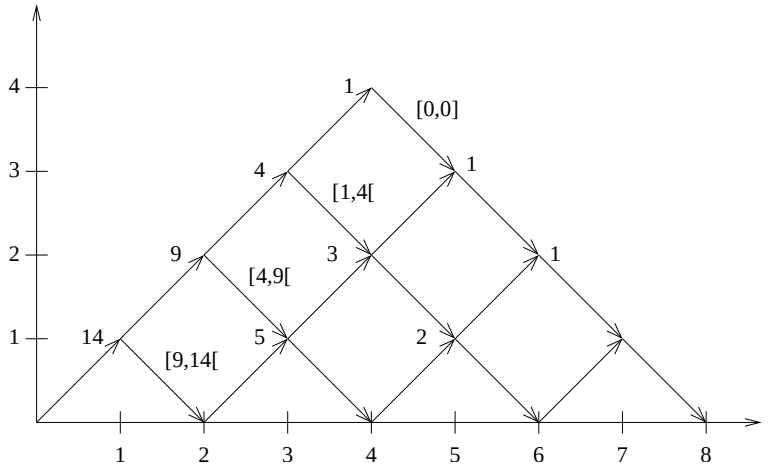


Figure 3.16: Paths

and proceed iteratively from $(i-1, j+1)$ by going up as long as possible and going down again. Remembering the number of parenthesis opened and closed along our way results in the required encoding. The following algorithm finalizes these ideas.

```
UnrankTree(n, r)
Input:  a number of inner nodes n and a rank r ∈ [0, C(n − 1)]
Output:  encoding of the inner leaves of a tree
lNoParOpen = 0;
lNoParClose = 0;
i = 1; // current encoding
j = 0; // current position in encoding array
while (j < n) {
  k = q(lNoParOpen + lNoParClose + 1, lNoParOpen - lNoParClose + 1);
  if (k ≤ r) {
    r -= k;
    ++lNoParClose;
  } else {
    aTreeEncoding[j++] = i;
    ++lNoParOpen;
  }
  ++i;
}
```

Given an array with the encoding of a tree, it is easy to construct the tree from it. The following procedure does that.

```
TreeEncoding2Tree(n, aEncoding) {
Input:  the number of internal nodes of the tree n
```

```
Output: root node of the result tree
  root = new Node; /* root of the result tree */
  curr = root; /* curr: current internal node whose subtrees are to be c
  i = 1; /* pointer to entry in encoding */
  child = 0; /* 0 = left , 1 = right: next child whose subtree is to be
  while (i < n) {
    lDiff = aEncoding[i] - aEncoding[i-1];
    for (k = 1; k < lDiff; ++k) {
      if (child == 0) {
        curr->addLeftLeaf();
        child = 1;
      } else {
        curr->addRightLeaf();
        while (curr->right() != 0) {
          curr = curr->parent();
        }
        child = 1;
      }
    }
    if (child == 0) {
      curr->left(new Node(curr)); // curr becomes parent of new node
      curr = curr->left();
      ++i;
      child = 0;
    } else {
      curr->right(new Node(curr));
      curr = curr->right();
      ++i;
      child = 0;
    }
  }
  while (curr != 0) {
    curr->addLeftLeaf(); // addLeftLeaf adds leaf if no left-child exists
    curr->addRightLeaf(); // analogous
    curr = curr->parent();
  }
return root;
}
```

3.3.3 Generating Random Join Trees without Cross Products

A general solution for randomly generating join trees without cross products is not known. However, if we restrict ourselves to acyclic queries, we can apply an algorithm developed by Galindo-Legaria, Pellenkoft, and Kersten [288, 287, 290]. For this algorithm to work, we have to assume that the query graph is connected and acyclic.

For the rest of this section, we assume that $G = (V, E)$ is the query graph and $|V| = n$. That is, n relations are to be joined. No join tree contains a cross product.

With every node in a join tree, we associate a *level*. The root has level 0. Its children have level 1, and so on. We further use lower-case letters for relations.

For a given query graph G, we denote by $\mathcal{T}_G$ the set of join trees for G. Let $\mathcal{T}_G^{v(k)} \subseteq \mathcal{T}_G$ be the subset of join trees where the leaf node (i.e. relation) v occurs at level k. Some trivial observations follow. If the query graph consists of a single node ($n = 1$), then $|\mathcal{T}_G| = |\mathcal{T}_G^{v(0)}| = 1$. If $n > 1$, the top node in the join tree is a join and not a relation. Hence, $|\mathcal{T}_G^{v(0)}| = 0$. Obviously, the maximum level that can occur in any join tree is $n-1$. Hence, $|\mathcal{T}_G^{v(k)}| = 0$ if $k \geq n$. Since the level at which a leaf node v occurs in some join tree is unique, we have $\mathcal{T}_G = \cup_{k=0}^{n} \mathcal{T}_G^{v(k)}$ and $\mathcal{T}_G^{v(i)} \cap \mathcal{T}_G^{v(j)} = \emptyset$ for $i \neq j$. This gives us $|\mathcal{T}_G| = \sum_{k=0}^{n} |\mathcal{T}_G^{v(k)}|$.

The algorithm generates an unordered tree with n leaf nodes. If we wish to have a random ordered tree, we have to pick one of the 2^{n-1} possibilities to order the $(n-1)$ joins within the tree. We proceed as follows. We start with some notation for lists, discuss how two lists can be merged, describe how a specific merge can be specified, and count the number of possible merges. This is important, since join trees will be described as lists of trees. Given a leaf node v, we simply traverse the path from the root to v. Thereby, subtrees that branch off can be collected into a list of trees. After these remarks, we start developing the algorithm in several steps. First, we consider two operations with which we can construct new join trees: *leaf-insertion* introduces a new leaf node into a given tree and *tree-merging* merges two join trees. Since we do not want to generate cross products in this section, we have to apply these operations carefully. Therefor, we need a description of how to generate *all* valid join trees for a given query graph. The central data structure for this purpose is the *standard decomposition graph* (SDG). Hence, in the second step, we define SDGs and introduce an algorithm that derives an SDG from a given query graph. In the third step, we start counting. The fourth and final step consists of the unranking algorithm. We do not discuss the ranking algorithm. It can be found in [290].

We use the Prolog notation $|$ to separate the first element of a list from its tail. For example, the list $\langle a|t\rangle$ has a as its first element and a tail t. Assume that P is a property of elements. A list l' is the *projection* of a list L on P, if L' contains all elements of L satisfying the property P. Thereby, the order is retained. A list L is a *merge* of two disjoint lists L_1 and L_2 if L contains all elements from L_1 and L_2 and both are projections of L.

A merge of a list L_1 with a list L_2 whose respective lengths are l_1 and l_2 can be described by an array $\alpha = [\alpha_0, \ldots, \alpha_{l_2}]$ of non-negative integers whose sum is equal to l_1. The non-negative integer α_{i-1} gives the number of elements of L_1 which precede the i-th element of L_2 in the merged list. We obtain the merged list L by first taking α_0 elements from L_1. Then, an element from L_2 follows. Then α_1 elements from L_1 and the next element of L_2 follow and so on. Finally follow the last α_{l_2} elements of L_1. Figure 3.17 illustrates possible merges.

Compare list merges to the problem of non-negative (weak) integer composition [**?**]. There, we ask for the number of compositions of a non-negative integer n into k non-negative integers α_i with $\sum_{i=1}^{k} \alpha_k = n$. The answer is $\binom{n+k-1}{k-1}$ [798]. Since we have to decompose l_1 into $l_2 + 1$ non-negative integers, the number of possible merges is $M(l_1, l_2) = \binom{l_1+l_2}{l_2}$. The observation $M(l_1, l_2) = M(l_1 - 1, l_2) + M(l_1, l_2 - 1)$

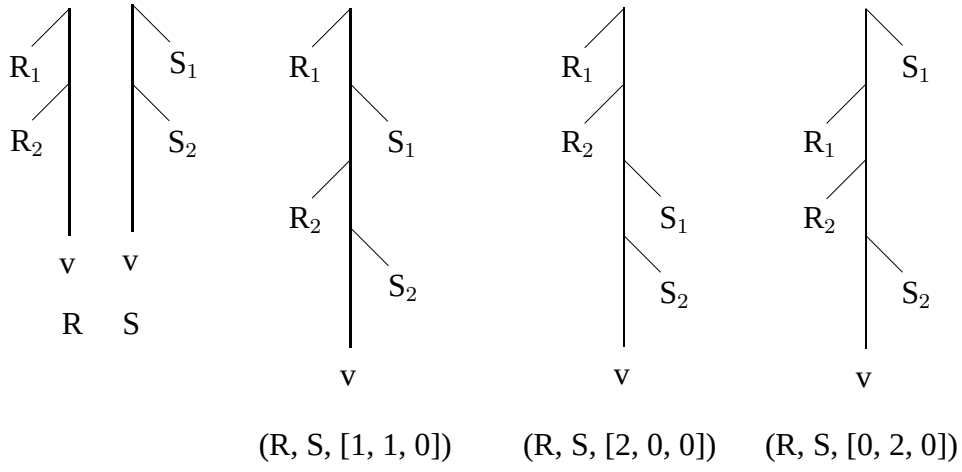


Figure 3.17: Tree-merge

allows us to construct an array of size $n * n$ in $O(n^2)$ that materializes the values for M. This array will allow us to rank list merges in $O(l_1 + l_2)$.

The idea for establishing a bijection between $[1, M(l_1, l_2)]$ and the possible αs is a general one and used for all subsequent algorithms of this section. Assume that we want to rank the elements of some set S and $S = \cup_{i=0}^{n} S_i$ is partitioned into disjoint S_i. If we want to rank $x \in S_k$, we first find the *local rank* of $x \in S_k$. The rank of x is then defined as

$$\sum_{i=0}^{k-1} |S_i| + \texttt{local-rank}(x, S_k)$$

To unrank some number $r \in [1, N]$, we first find k such that

$$k = \min_j (r \leq \sum_{i=0}^{j} |S_i|)$$

Then, we proceed by unranking with the new local rank

$$r' = r - \sum_{i=0}^{k-1} |S_i|$$

within S_k.

Accordingly, we partition the set of all possible merges into subsets. Each subset is determined by α_0. For example, the set of possible merges of two lists L_1 and L_2 with length $l_1 = l_2 = 4$ is partitioned into subsets with $\alpha_0 = j$ for $0 \leq j \leq 4$. In each partition, we have $M(j, l_2 - 1)$ elements. To unrank a number $r \in [1, M(l_1, l_2)]$, we first determine the partition by computing $k = \min_j r \leq \sum_{i=0}^{j} M(j, l_2 - 1)$. Then, $\alpha_0 = l_1 - k$. With the new rank $r' = r - \sum_{i=0}^{k} M(j, l_2 - 1)$, we start iterating all over. The following table gives the numbers for our example and can be used to understand the unranking algorithm. The algorithm itself can be found in Figure 3.18.

k	α_0	$(k, l_2 - 1)$	$M(k, l_2 - 1)$	rank intervals
0	4	$(0, 3)$	1	$[1, 1]$
1	3	$(1, 3)$	4	$[2, 5]$
2	2	$(2, 3)$	10	$[6, 15]$
3	1	$(3, 3)$	20	$[16, 35]$
4	0	$(4, 3)$	35	$[36, 70]$

```
UnrankDecomposition(r, l1, l2)
Input:  a rank r, two list sizes l1 and l2
Output:  a merge specification α.
for (i = 0; i ≤ l2; ++i) {
  alpha[i] = 0;
}
i = k = 0;
while (l1 > 0 && l2 > 0) {
  m = M(k, l2 − 1);
  if (r ≤ m) {
    alpha[i++] = l1 − k;
    l1 = k;
    k = 0;
    −−l2;
  } else {
    r− = m;
    ++k;
  }
}
alpha[i] = l1;
return alpha;
```

Figure 3.18: Algorithm `UnrankDecomposition`

We now turn to the *anchored list representation* of join trees.

Definition 3.3.1 *Let T be a join tree and v be a leaf of T. The* anchored list representation *L of T is constructed as follows:*

- *If T consists of the single leaf node v, then $L = \langle\rangle$.*
- *If $T = (T_1 \bowtie T_2)$ and without loss of generality v occurs in T_2, then $L = \langle T_1 | L_2 \rangle$, where L_2 is the anchored list representation of T_2.*

We then write $T = (L, v)$.

Observe that if $T = (L, v) \in \mathcal{T}_G$, then $T \in \mathcal{T}_G^{v(k)} \prec\!\!\succ |L| = k$.

The operation *leaf-insertion* is illustrated in Figure 3.19. A new leaf v is inserted into the tree at level k. Formally, it is defined as follows.

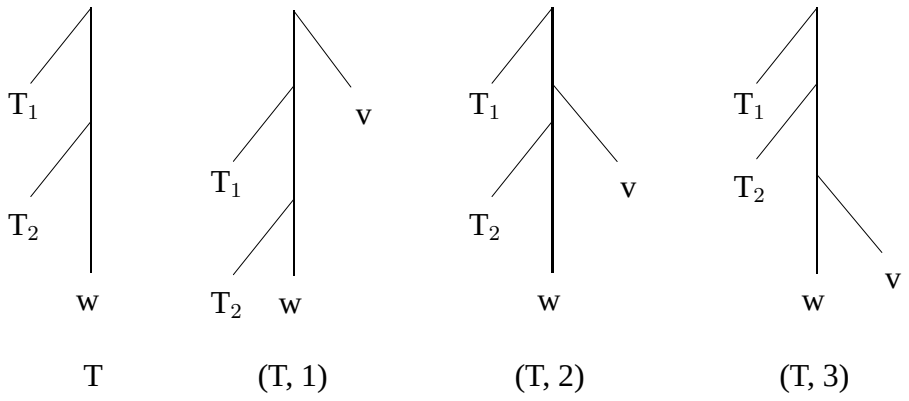


Figure 3.19: Leaf-insertion

Definition 3.3.2 *Let $G = (V, E)$ be a query graph, T a join tree of G. $v \in V$ be such that $G' = G|_{V \setminus \{v\}}$ is connected, $(v, w) \in E$, $1 \leq k < n$, and*

$$T = (\langle T_1, \ldots, T_{k-1}, v, T_{k+1}, \ldots, T_n \rangle, w) \quad (3.10)$$
$$T' = (\langle T_1, \ldots, T_{k-1}, T_{k+1}, \ldots, T_n \rangle, w). \quad (3.11)$$

Then we call (T', k) an insertion pair *on v and say that T is* decomposed into *(or* constructed from*) the pair (T', k) on v.*

Observe that leaf-insertion defines a bijective mapping between $\mathcal{T}_G^{v(k)}$ and insertion pairs (T', k) on v, where T' is an element of the disjoint union $\cup_{i=k-1}^{n-2} \mathcal{T}_{G'}^{w(i)}$.

The operation *tree-merging* is illustrated in Figure 3.17. Two trees $R = (L_R, w)$ and $S = (L_S, w)$ on a common leaf w are merged by merging their anchored list representations.

Definition 3.3.3 *Let $G = (V, E)$ be a query graph, $w \in V$, $T = (L, w)$ a join tree of G, $V_1, V_2 \subseteq V$ such that $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{w\}$. For $i = 1, 2$:*

- *Define the property P_i to be "every leaf of the subtree is in V_i",*
- *Let L_i be the projection of L on P_i.*
- *$T_i = (L_i, w)$.*

Let α be the integer composition such that L is the result of merging L_1 and L_2 on α. Then we call (T_1, T_2, α) a merge triplet*. We say that T is* decomposed into *(*constructed from*) (T_1, T_2, α) on V_1 and V_2.*

Observe that the *tree-merging* operation defines a bijective mapping between $\mathcal{T}_G^{w(k)}$ and merge triplets (T_1, T_2, α), where $T_1 \in \mathcal{T}_{G_1}^{w(i)}$, $T_2 \in \mathcal{T}_{G_2}^{w(k-i)}$, and α specifies a merge of two lists of sizes i and $k - i$. Further, the number of these merges (i.e. the number of possibilities for α) is $\binom{i+(k-i)}{k-i} = \binom{k}{i}$.

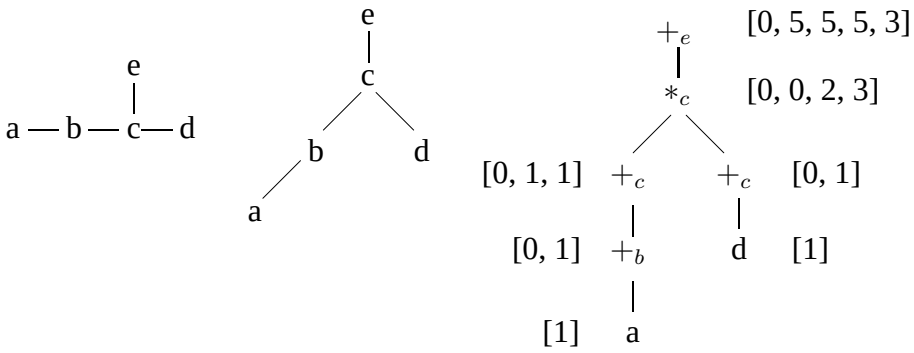


Figure 3.20: A query graph, its tree, and its standard decomposition graph

A *standard decomposition graph* of a query graph describes the possible constructions of join trees. It is not unique (for $n > 1$) but anyone can be used to construct all possible unordered join trees. For each of our two operations it has one kind of inner nodes. A unary node labeled $+_v$ stands for leaf-insertion of v. A binary node labeled $*_w$ stands for tree-merging its subtrees whose only common leaf is w.

The standard decomposition graph of a query graph $G = (V, E)$ is constructed in three steps:

1. pick an arbitrary node $r \in V$ as its root node;
2. transform G into a tree G' by directing all edges away from r;
3. call `QG2SDG(`G', r`)`

with

```
QG2SDG(G', r)
Input:  a query tree G' = (V, E) and its root r
Output:  a standard query decomposition tree of G'
Let {w_1, ..., w_n} be the children of v;
switch (n) {
  case 0:  label v with "v";
  case 1:
       label v as "+_v";
       QG2SDG(G', w_1);
  otherwise:
       label v as "*_v";
       create new nodes l, r with label +_v;
       E \=  {(v, w_i)|1 ≤ i ≤ n};
       E ∪=  {(v, l), (v, r), (l, w_1)} ∪ {(r, w_i)|2 ≤ i ≤ n};
       QG2SDG(G', l);
       QG2SDG(G', r);
}
return G';
```

Note that `QG2SDG` transforms the original graph G' into its SDG by side-effects. Thereby, the n-ary tree is transformed into a binary tree similar to the procedure described by Knuth [492, Chap 2.3.2]. Figure 3.20 shows a query graph G, its tree G' rooted at e, and its standard decomposition tree.

For an efficient access to the number of join trees in some partition $\mathcal{T}_G^{v(k)}$ in the unranking algorithm, we materialize these numbers. This is done in the `count` array. The semantics of a `count` array $[c_0, c_1, \ldots, c_n]$ of a node u with label $\circ_v$ ($\circ \in \{+, *\}$) of the SDG is that u can construct c_i different trees in which leaf v is at level i. Then, the total number of trees for a query can be computed by summing up all the c_i in the `count` array of the root node of the decomposition tree.

To compute the `count` and an additional `summand` adornment of a node labeled $+_v$, we use the following lemma.

Lemma 3.3.4 *Let $G = (V, E)$ be a query graph with n nodes, $v \in V$ such that $G' = G|_{V \setminus v}$ is connected, $(v, w) \in E$, and $1 \leq k < n$. Then*

$$|\mathcal{T}_G^{v(k)}| = \sum_{i \geq k-1} |\mathcal{T}_{G'}^{w(i)}|$$

This lemma follows from the observation made after the definition of the leaf-insertion operation.

The sets $\mathcal{T}_{G'}^{w(i)}$ used in the summands of Lemma 3.3.4 directly correspond to subsets $\mathcal{T}_G^{v(k),i}$ ($k - 1 \leq i \leq n - 2$) defined such that $T \in \mathcal{T}_G^{v(k),i}$ if

1. $T \in \mathcal{T}_G^{v(k)}$,
2. the insertion pair on v of T is (T', k), and
3. $T' \in \mathcal{T}_{G'}^{w(i)}$.

Further, $|\mathcal{T}_G^{v(k),i}| = |\mathcal{T}_{G'}^{w(i)}|$. For efficiency, we materialize the summands in an array of arrays `summands`.

To compute the `count` and `summand` adornment of a node labeled $*_v$, we use the following lemma.

Lemma 3.3.5 *Let $G = (V, E)$ be a query graph, $w \in V$, $T = (L, w)$ a join tree of G, $V_1, V_2 \subseteq V$ such that $G_1 = G|_{V_1}$ and $G_2 = G|_{V_2}$ are connected, $V_1 \cup V_2 = V$, and $V_1 \cap V_2 = \{v\}$. Then*

$$|\mathcal{T}_G^{v(k)}| = \sum_i \binom{k}{i} |\mathcal{T}_{G_1}^{v(i)}| \, |\mathcal{T}_{G_2}^{v(k-i)}|$$

This lemma follows from the observation made after the definition of the tree-merge operation.

The sets $\mathcal{T}_{G'}^{w(i)}$ used in the summands of Lemma 3.3.5 directly correspond to subsets $\mathcal{T}_G^{v(k),i}$ ($0 \leq i \leq k$) defined such that $T \in \mathcal{T}_G^{v(k),i}$ if

1. $T \in \mathcal{T}_G^{v(k)}$,

2. the merge triplet on V_1 and V_2 of T is (T_1, T_2, α), and

3. $T_1 \in \mathcal{T}_{G_1}^{v(i)}$.

Further, $|\mathcal{T}_G^{v(k),i}| = \binom{k}{i} |\mathcal{T}_{G_1}^{v(i)}| \, |\mathcal{T}_{G_2}^{v(k-i)}|$.

Before we come to the algorithm for computing the adornments `count` and `summands`, let us make one observation that follows directly from the above two lemmata. Assume a node v whose `count` array is $[c_1, \ldots, c_m]$ and whose `summands` is $s = [s^0, \ldots, s^n]$ with $s_i = [s_0^i, \ldots, s_m^i]$, then $c_i = \sum_{j=0}^{m} s_j^i$ holds. Figure 3.21 contains the algorithm to adorn SDG's nodes with `count` and `summands`. It has worst-case complexity $O(n^3)$. Figure 3.20 shows the `count` adornment for the SDG. Looking at the `count` array of the root node, we see that the total number of join trees for our example query graph is 18.

`Adorn(`v`)`
Input: a node v of the SDG
Output: v and nodes below are adorned by count and summands
Let $\{w_1, \ldots, w_n\}$ be the children of v;
switch (n) {
  case 0: count(v) := $[1]$; // no summands for v
  case 1:
    Adorn(w_1);
    assume count(w_1) = $[c_0^1, \ldots, c_{m_1}^1]$;
    count(v) = $[0, c_1, \ldots, c_{m_1+1}]$ where $c_k = \sum_{i=k-1}^{m_1} c_i^1$;
    summands(v) = $[s^0, \ldots, s^{m_1+1}]$ where $s^k = [s_0^k, \ldots, s_{m_1+1}^k]$ and
    $$s_i^k = \begin{cases} c_i^1 & \text{if } 0 < k \text{ and } k-1 \leq i \\ 0 & \text{else} \end{cases}$$
  case 2:
    Adorn(w_1);
    Adorn(w_2);
    assume count(w_1) = $[c_0^1, \ldots, c_{m_1}^1]$;
    assume count(w_2) = $[c_0^2, \ldots, c_{m_2}^2]$;
    count(v) = $[c_0, \ldots, c_{m_1+m_2}]$ where
      $c_k = \sum_{i=0}^{m_1} \binom{k}{i} c_i^1 c_{k-i}^2$; // $c_i^2 = 0$ for $i \notin \{0, \ldots, m_2\}$
    summands(v) = $[s^0, \ldots, s^{m_1+m_2}]$ where $s^k = [s_0^k, \ldots, s_{m_1}^k]$ and
    $$s_i^k = \begin{cases} \binom{k}{i} c_i^1 c_{k-i}^2 & \text{if } 0 \leq k-i \leq m_2 \\ 0 & \text{else} \end{cases}$$
}

Figure 3.21: Algorithm `Adorn`

The algorithm `UnrankLocalTreeNoCross` called by `UnrankTreeNoCross` adorns the standard decomposition graph with `insert-at` and `merge-using` annotations. These can then be used to extract the join tree.

```
UnrankTreeNoCross(r,v)
Input: a rank r and the root v of the SDG
Output: adorned SDG
let count(v) = [x_0, ..., x_m];
k := min_j r ≤ Σ_{i=0}^{j} x_i; // efficiency: binary search on materialized sums.
r' := r − Σ_{i=0}^{k−1} x_i;
UnrankLocalTreeNoCross(v, r', k);
```

The following table shows the intervals associated with the partitions $\mathcal{T}_G^{e(k)}$ for the standard decomposition graph in Figure 3.20:

Partition	Interval
$\mathcal{T}_G^{e(1)}$	$[1,5]$
$\mathcal{T}_G^{e(2)}$	$[6,10]$
$\mathcal{T}_G^{e(3)}$	$[11,15]$
$\mathcal{T}_G^{e(4)}$	$[16,18]$

The unranking procedure makes use of unranking decompositions and unranking triples. For the latter and a given X, Y, Z, we need to assign each member in

$$\{(x,y,z)|1 \leq x \leq X, 1 \leq y \leq Y, 1 \leq z \leq Z\}$$

a unique number in $[1, XYZ]$ and base an unranking algorithm on this assignment. We leave this as a simple exercise to the reader and call the function `UnrankTriplet`(r, X, Y, Z). Here, r is the rank and X, Y, and Z are the upper bounds for the numbers in the triplets. The code for unranking looks as follows:

```
UnrankingTreeNoCrossLocal(v, r, k)
Input: an SDG node v, a rank r, a number k identifying a partition
Output: adornments of the SDG as a side-effect
Let {w_1, ..., w_n} be the children of v
switch (n) {
  case 0:
    assert(r = 1 && k = 0);
    // no additional adornment for v
  case 1:
    let count(v) = [c_0, ..., c_n];
    let summands(v) = [s^0, ..., s^n];
    assert(k ≤ n && r ≤ c_k);
    k_1 = min_j r ≤ Σ_{i=0}^{j} s_i^k;
    r_1 = r − Σ_{i=0}^{k_1−1} s_i^k;
    insert-at(v) = k;
    UnrankingTreeNoCrossLocal(w_1, r_1, k_1);
  case 2:
    let count(v) = [c_0, ..., c_n];
```

```
    let summands(v) = [s^0, ..., s^n];
    let count(w_1) = [c^1_0, ..., c^1_{n_1}];
    let count(w_2) = [c^2_0, ..., c^2_{n_2}];
    assert(k ≤ n  &&  r ≤ c_k);
    k_1 = min_j r ≤ Σ_{i=0}^{j} s^k_i;
    q = r − Σ_{i=0}^{k_1−1} s^k_i;
    k_2 = k − k_1;
    (r_1, r_2, a) = UnrankTriplet(q, c^1_{k_1}, c^2_{k_2}, (k choose i));
    α = UnrankDecomposition(a);
    merge-using(v) = α;
    UnrankingTreeNoCrossLocal(w_1, r_1, k_1);
    UnrankingTreeNoCrossLocal(w_2, r_2, k_2);
}
```

3.3.4 Quick Pick

The QuickPick algorithm of Waas and Pellenkoft [869, 870] does not generate random join trees in the strong sense but comes close to it and is far easier to implement and more broadly applicable. The idea is to randomly select an edge in the query graph and to construct a join tree corresponding to this edge.

```
QuickPick(Query Graph G)
Input:  a query graph G = ({R_1, ..., R_n}, E)
Output:  a bushy join tree
BestTreeFound = any join tree
while stopping criterion not fulfilled {
  E' = E;
  Trees = {R_1, ..., R_n};
  while (|Trees| > 1) {
    choose e ∈ E';
    E' −= e;
    if (e connects two relations in different subtrees T_1, T_2 ∈ Trees) {
      Trees -= T_1;
      Trees -= T_2;
      Trees += CreateJoinTree(T_1, T_2);
    }
  }
  Tree = single tree contained in Trees;
  if (cost(Tree) < cost(BestTreeFound)) {
    BestTreeFound = Tree;
  }
}
return BestTreeFound
```

3.3.5 Iterative Improvement

Swami and Gupta [829], Swami [828] and Ioannidis and Kang [438] applied the idea of iterative improvement to join ordering [438]. The idea is to start from a random plan and then to apply randomly selected transformations from a rule set if they improve the current join tree, until not further improvement is possible.

```
IterativeImprovementBase(Query Graph G)
Input:  a query graph G = ({R_1, ..., R_n}, E)
Output:  a join tree
do {
  JoinTree = random tree
  JoinTree = IterativeImprovement(JoinTree)
  if (cost(JoinTree) < cost(BestTree)) {
    BestTree = JoinTree;
  }
} while (time limit not exceeded)
return BestTree

IterativeImprovement(JoinTree)
Input:  a join tree
Output:  improved join tree
do {
  JoinTree' = randomly apply a transformation to JoinTree;
  if (cost(JoinTree') < cost(JoinTree)) {
    JoinTree = JoinTree';
  }
} while (local minimum not reached)
return JoinTree
```

The number of variants of iterative improvements is large. The first parameter is the used rule set. To restrict search to left-deep trees, a rule set consisting of *swap* and *3cycle* is appropriate [829]. If we consider bushy trees, a complete set consisting of commutativity, associativity, left join exchange and right join exchange makes sense. This rule set (proposed by Ioannidis and Kang) is appropriate to explore the whole space of bushy join trees. A second parameter is how to determine whether the local minimum has been reached. Considering all possible neighbor states of a join tree is expensive. Therefor, a subset of size k is sometimes considered. Then, for example, k can be limited to the number of edges in the query graph [829].

3.3.6 Simulated Annealing

Iterative Improvement suffers from the drawback that it only applies a move if it improves the current plan. This leads to the problem that one is often stuck in a local minimum. Simulated annealing tries to avoid this problem by allowing moves that

result in more expensive plans [443, 438, 829]. However, instead of considering every plan, only those whose cost increase does not exceed a certain limit are considered. During time, this limit decreases. This general idea is cast into the notion temperatures and probabilities of performing a selected transformation. A generic formulation of simulated annealing could look as follows:

```
SimulatedAnnealing(Query Graph G)
Input:  a query graph G = ({R_1, ..., R_n}, E)
Output:  a join tree
BestTreeSoFar = random tree;
Tree = BestTreeSoFar;
do {
  do {
    Tree' = apply random transformation to Tree;
    if (cost(Tree') < cost(Tree)) {
      Tree = Tree';
    } else {
      with probability e^{-(cost(Tree') - cost(Tree))/temperature}
        Tree = Tree';
    }
    if (cost(Tree) < cost(BestTreeSoFar)) {
      BestTreeSoFar = Tree';
    }
  } while (equilibrium not reached)
reduce temperature;
} while (not frozen)
return BestTreeSoFar
```

Besides the rule set used, the initial temperature, the temperature reduction, and the definitions of equilibrium and frozen determine the algorithm's behavior. For each of them several alternatives have been proposed in the literature. The starting temperature can be calculated as follows: determine the standard deviation σ of costs by sampling and multiply it with a constant value ([829] use 20). An alternative is to set the starting temperature twice the cost of the first randomly selected join tree [438] or to determine the starting temperature such that at least 40% of all possible transformations are accepted [803].

For temperature reduction, we can apply the formula $temp* = 0.975$ [438] or $\max(0.5, e^{-\frac{\lambda t}{\sigma}})$ [829].

The equilibrium is defined to be reached if for example the cost distribution of the generated solutions is sufficiently stable [829], the number of iterations is sixteen times the number of relations in the query [438], or number of iterations is the same as the number of relations in the query [803].

We can establish frozenness if the difference between the maximum and minimum costs among all accepted join trees at the current temperature equals the maximum change in cost in any accepted move at the current temperature [829], the current

solution could not be improved in four outer loop iterations and the temperature has been fallen below one [438], or the current solution could not be improved in five outer loop iterations and less than two percent of the generated moves were accepted [803].

Considering databases are used in mission critical applitions. Would you bet your business on these numbers?

3.3.7 Tabu Search

Morzy, Matyasiak and Salza applied Tabu Search to join ordering [606]. The general idea is that among all neighbors reachable via the transformations, only the cheapest is considered even if its cost are higher than the costs of the current join tree. In order to avoid running into cycles, a tabu set is maintained. It contains the last join trees generated, and the algorithm is not allowed to visit them again. This way, it can escape local minima, since eventually all nodes in the valley of a local minimum will be in the tabu set.

Tabu Search looks as follows:

```
TabuSearch(Query Graph)
Input: a query graph G = ({R_1, ..., R_n}, E)
Output: a join tree
Tree = random join tree;
BestTreeSoFar = Tree;
TabuSet = ∅;
do
  Neighbors = all trees generated by applying a transformation to Tree;
  Tree = cheapest in Neighbors \ TabuSet;
  if (cost(Tree) < cost(BestTreeSoFar)) {
    BestTreeSoFar = Tree;
  }
  if(|TabuSet| > limit) remove oldest tree from TabuSet;
  TabuSet += Tree;
return BestTreeSoFar;
```

3.3.8 Genetic Algorithms

Genetic algorithms are inspired by evolution: only the fittest survives [325]. They work with a population that evolves from generation to generation. Successors are generated by crossover and mutation. Further, a subset of the current population (the fittest) are propagated to the next generation (selection). The first generation is generated by a random generation process.

The problem is how to represent each individual in a population. The following analogies are used:

- Chromosome $\longleftrightarrow$ string
- Gene $\longleftrightarrow$ character

In order to solve an optimization problem with genetic algorithms, an encoding is needed as well as a specification for selection, crossover, and mutation.

Genetic algorithms for join ordering have been considered in [70, 803]. We first introduce alternative encodings, then come to the selection process, and finally discuss crossover and mutation.

Encodings We distinguish *ordered list* and *ordinal number* encodings. Both encodings are used for left-deep and bushy trees. In all cases we assume that the relations $R_1, \ldots, R_n$ are to be joined and use the index i to denote R_i.

1. Ordered List Encoding

   (a) left-deep trees
   A left-deep join tree is encoded by a permutation of $1, \ldots, n$. For instance, $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1423".

   (b) bushy trees
   Bennet, Ferris, and Ioannidis proposed the following encoding scheme [70, 71]. A bushy join-tree without cartesian products is encoded as an ordered list of the edges in the join graph. Therefore, we number the edges in the join graph. Then the join tree is encoded in a bottom-up, left-to-right manner. See Figure 3.22 for an example.

2. Ordinal Number Encoding

   (a) left-deep trees
   A join tree is encoded by using a list of relations that is shortened whenever a join has been encoded. We start with the list $L = \langle R_1, \ldots, R_n \rangle$. Then within L we find the index of first relation to be joined. Let this relation be R_i. R_i is the i-th relation in L. Hence, the first character in the chromosome string is i. We eliminate R_i from L. For every subsequent relation joined, we again determine its index in L, remove it from L and append the index to the chromosome string. For instance, starting with $\langle R_1, R_2, R_3, R_4 \rangle$, the left-deep join tree $(((R_1 \bowtie R_4) \bowtie R_2) \bowtie R_3)$ is encoded as "1311".

   (b) bushy trees
   Again, we start with the list $L = \langle R_1, \ldots, R_n \rangle$ and encode a bushy join tree in a bottom-up, left-to-right manner. Let $R_i \bowtie R_j$ be the first join in the join tree under this ordering. Then we look up their positions in L and add them to the encoding. Next we eliminate R_i and R_j from L and push $R_{i,j}$ to the front of it. We then proceed for the other joins by again selecting the next join which now can be between relations and/or subtrees. We determine their position within L, add these positions to the encoding, remove them from L, and insert a composite relation into L such that the new composite relation directly follows those already present. For instance, starting with the list $\langle R_1, R_2, R_3, R_4 \rangle$, the bushy join tree $((R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4))$ is encoded as "12 23 12".

The encoding is completed by adding join methods.

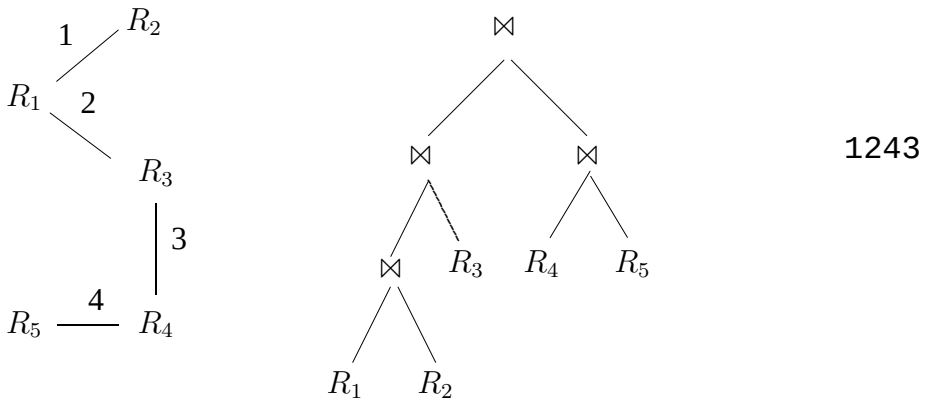


Figure 3.22: A query graph, a join tree, and its encoding

Crossover A crossover generates a new solution from two individuals. Therefore, two partial solutions are combined. Obviously, its definition depends on the encoding. Two kinds of crossovers are distinguished: the subsequence and the subset exchange.

The subsequence exchange for the ordered list encoding works as follows. Assume two individuals with chromosomes $u_1v_1w_1$ and $u_2v_2w_2$. From these we generate $u_1v_1'w_1$ and $u_2v_2'w_2$, where v_i' is a permutation of the relations in v_i such that the order of their appearence is the same as in $u_{3-i}v_{3-i}w_{3-i}$. In order to adapt the subsequence exchange operator to the ordinal number encoding, we have to require that the v_i are of equal length ($|v_1| = |v_2|$) and occur at the same offset ($|u_1| = |u_2|$). We then simply swap the v_i. That is, we generate $u_1v_2w_1$ and $u_2v_1w_2$.

The subset exchange is defined only for the ordered list encoding. Within the two chromosomes, we find two subsequences of equal length comprising the same set of relations. These sequences are then simply exchanged.

Mutation A mutation randomly alters a character in the encoding. If duplicates must not occur — as in the ordered list encoding — swapping two characters is a perfect mutation.

Selection The probability of a join tree's survival is determined by its rank in the population. That is, we calculate the costs of the join trees encoded for each member of the population. Then we sort the population according to their associated costs and assign probabilities to each individual such that the best solution in the population has the highest probability to survive and so on. After probabilities have been assigned, we randomly select members of the population taking these probabilities into account. That is, the higher the probability of a member, the higher is its chance to survive.

Algorithm The genetic algorithm then works as follows. First, we create a random population of a given size (say 128). We apply crossover and mutation with a given rate, for example such that 65% of all members of a population participate in crossover, and 5% of all members of a population are subject to random mutation. Then we apply selection until we again have a population of a given size. We stop after we have not seen an improvement within the population for a fixed number of iterations (say 30).

3.4 Hybrid Algorithms

All the algorithms we have seen so far can be combined to result in new approaches to join ordering. Some of the numerous possibilities have been described in the literature. We present them.

3.4.1 Two Phase Optimization

Two phase optimization combines Iterative Improvement with Simulated Annealing [438]. For a number of randomly generated initial trees, Iterative Improvement is used to find a local minimum. Then Simulated Annealing is started to find a better plan in the neighborhood of the local minima. The initial temperature of Simulated Annealing can be lower as is its original variants.

3.4.2 AB-Algorithm

The AB-Algorithm was developed by Swami and Iyer [830, 831]. It builds on the IKKBZ-Algorithm by resolving its limitations. First, if the query graph is cyclic, a spanning tree is selected. Second, two different cost functions for joins (join methods) are supported by the AB-Algorithm: nested loop join and sort merge join. In order to make the sort merge join's cost model fit the ASI property, it is simplified. Third, join methods are assigned randomly before IKKBZ is called. Afterwards, an iterative improvement phase follows. The algorithm can be formulated as follows:

```
AB(Query Graph G)
Input:  a query graph G = ({R_1, ..., R_n}, E)
Output:  a left-deep join tree
while (number of iterations ≤ n^2) {
  if G is cyclic take spanning tree of G
  randomly attach a join method to each relation
  JoinTree = result of IKKBZ
  while (number of iterations ≤ n^2) {
    apply Iterative Improvement to JoinTree
  }
}
return best tree found
```

3.4.3 Toured Simulated Annealing

Lanzelotte, Valduriez, and Zäit introduced *toured simulated annealing* as a search strategy useful in distributed databases where the search space is even larger than in centralized systems [515]. The basic idea is that simulated annealing is called n times with different initial join trees, if n is the number of relations to be joined. Each join sequence in the set `Solutions` produced by `GreedyJoinOrdering-3` is used to start an independent run of simulated annealing. As a result, the starting temperature can be decreased to 0.1 times the cost of the initial plan.

3.4.4 GOO-II

GOO-II appends an Iterative Improvement step to the GOO-Algorithm.

3.4.5 Iterative Dynamic Programming

Iterative Dynamic Programming combines heuristics with dynamic programming in order to overcome the deficiencies of both. It comes in two variants [499, 775]. The first variant, `IDP-1` (see Figure 3.23), first creates all join trees which contain up to k relations where k is a parameter of the algorithm. After this step, it selects the cheapest join tree comprising k relations, replaces it by a new compound relation and starts all over again. The iteration stops, when only one compound relation representing a join tree for all relations remains in the `ToDo` list.

The second variant, `IDP-2` (see Figure 3.24), works the other way round. It first applies a greedy heuristics to build join trees of size up to k. To the larger subtree it applies dynamic programming to improve it. The result of the optimized outcome of the greedy algorithm is then encapsulated in a new compound relation which replaces its constituent relations in the `ToDo` list. The algorithm then iterates until only one entry remains in the `ToDo` list.

Obviously, from these two basic variants several others can be derived. A systematic investigation of the basic algorithms and their variants is given by Kossmann and Stocker [499]. It turns out that the most promising variants exist for `IDP-1`.

3.5 Ordering Order-Preserving Joins

This section covers an algorithm for ordering order-preserving joins [599]. This is important for XQuery and other languages that require order-preservation. XQuery specifies that the result of a query is a sequence. If no `unordered` or `order by` instruction is given, the order of the output sequence is determined by the order of the input sequences given in the `for` clauses of the query. If there are several entries in a `for` clause or several `for` clauses, order-preserving join operators [180] can be a natural component for the evaluation of such a query.

The order-preserving join operator is used in several algebras in the context of

- semi-structured data and XML (e.g. SAL [66], XAL [278]),
- OLAP [790], and
- time series data [528].

We give a polynomial algorithm that produces bushy trees for a sequence of order-preserving joins and selections. These trees may contain cross products even if the join graph is connected. However, we apply selections as early as possible. The algorithm then produces the optimal plan among those who push selections down. The cost function is a parameter of the algorithm, and we do not need to restrict ourselves to those having the ASI property. Further, we need no restriction on the join graph, i.e. the algorithm produces the optimal plan even if the join graph is cyclic.

Before defining the order-preserving join, we need some preliminaries. The above algebras work on sequences of sets of variable bindings, i.e. sequences of unordered

```
IDP-1({R_1, ..., R_n}, k)
Input: a set of relations to be joined, maximum block size k
Output: a join tree
for (i = 1; i <= n; ++i) {
  BestTree({R_i}) = R_i;
}
ToDo = {R_1, ..., R_n};
while (|ToDo| > 1) {
  k = min(k, |ToDo|);
  for (i = 2; i < k; ++i) {
    for all S ⊆ ToDo, |S| = i do {
      for all O ⊂ S do {
        BestTree(S) = CreateJoinTree(BestTree(S), BestTree(O));
      }
    }
  }
  find V ⊂ ToDo, |V| = k
  with cost(BestTree(V)) = min{cost(BestTree(W)) | W ⊂ ToDo, |W| = k};
  generate new symbol T;
  BestTree({T}) = BestTree(V);
  ToDo = (ToDo \ V) ∪ {T};
  for all O ⊂ V do delete(BestTree(O));
}
return BestTree({R_1, ..., R_n});
```

Figure 3.23: Pseudo code for `IDP-1`

tuples where every attribute corresponds to a variable. (See Chapter 6.3 for a general discussion.) Single tuples are constructed using the standard $[\cdot]$ brackets. Concatenation of tuples and functions is denoted by $\circ$. The set of attributes defined for an expression e is defined as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$. For sequences e, we use $\alpha(e)$ to denote the first element of a sequence. We identify single element sequences with elements. The function τ retrieves the tail of a sequence, and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ.

We define the algebraic operators recursively on their input sequences. The order-preserving join operator is defined as the concatenation of an order-preserving selection and an order-preserving cross product. For unary operators, if the input sequence is empty, the output sequence is also empty. For binary operators, the output sequence is empty whenever the left operand represents an empty sequence.

The order-preserving join operator is based on the definition of an order-preserving cross product operator defined as

$$e_1 \hat{\times} e_2 := (\alpha(e_1) \hat{\overline{\times}} e_2) \oplus (\tau(e_1) \hat{\times} e_2)$$

```
IDP-2({R_1, ..., R_n}, k)
Input: a set of relations to be joined, maximum block size k
Output: a join tree
for (i = 1; i <= n; ++i) {
  BestTree({R_i}) = R_i;
}
ToDo = {R_1, ..., R_n};
while (|ToDo| > 1) {
  // apply greedy algorithm to select a good building block
  B = ∅;
  for all v ∈ ToDo, do {
    B += BestTree({v});
  }
  do {
    find L, R ∈ B
    with cost(CreateJoinTree(L,R))
         = min{cost(CreateJoinTree(L',R')) | L', R' ∈ B};
    P = CreateJoinTree(L,R));
    B = (B \ {L,R}) ∪ {P};
  } while (P involves no more than k relations and |B| > 1);
  // reoptimize the bigger of L and R,
  // selected in the last iteration of the greedy loop
  if (L involves more tables than R) {
    ReOpRels = relations involved in L;
  } else {
    ReOpRels = relations involved in R;
  }
  P = DP-Bushy(ReOpRels);
  generate new symbol T;
  BestTree({T}) = P;
  ToDo = (ToDo \ ReOpRels) ∪ {T};
  for all O ⊂ V do delete(BestTree(O));
}
return BestTree({R_1, ..., R_n});
```

Figure 3.24: Pseudocode for IDP-2

where

$$e_1 \hat{\overline{\times}} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \hat{\overline{\times}} \tau(e_2)) & \text{else} \end{cases}$$

We are now prepared to define the join operation on ordered sequences:

$$e_1 \hat{\bowtie}_p e_2 := \hat{\sigma}_p(e_1 \hat{\times} e_2)$$

where the order-preserving selection is defined as

$$\hat{\sigma}_p(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ \alpha(e) \oplus \hat{\sigma}_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \hat{\sigma}_p(\tau(e)) & \text{else} \end{cases}$$

As usual, selections can be reordered and pushed inside order-preserving joins. Besides, the latter are associative. The following equivalences formalize this.

$$\begin{array}{lcll} \hat{\sigma}_{p_1}(\hat{\sigma}_{p_2}(e)) & = & \hat{\sigma}_{p_2}(\hat{\sigma}_{p_1}(e)) & \\ \hat{\sigma}_{p_1}(e_1 \hat{\bowtie}_{p_2} e_2) & = & \hat{\sigma}_{p_1}(e_1) \hat{\bowtie}_{p_2} e_2 & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_1) \\ \hat{\sigma}_{p_1}(e_1 \hat{\bowtie}_{p_2} e_2) & = & e_1 \hat{\bowtie}_{p_2} \hat{\sigma}_{p_1}(e_2) & \text{if } \mathcal{F}(p_1) \subseteq \mathcal{A}(e_2) \\ e_1 \hat{\bowtie}_{p_1} (e_2 \hat{\bowtie}_{p_2} e_3) & = & (e_1 \hat{\bowtie}_{p_1} e_2) \hat{\bowtie}_{p_2} e_3 & \text{if } \mathcal{F}(p_i) \subseteq \mathcal{A}(e_i) \cup \mathcal{A}(e_{i+1}) \end{array}$$

While being associative, the order-preserving join is not commutative, as the following example illustrates. Given two tuple sequences $R_1 = \langle [a:1], [a:2] \rangle$ and $R_2 = \langle [b:1], [b:2] \rangle$, we have

$$\begin{array}{rcl} R_1 \hat{\bowtie}_{true} R_2 & = & \langle [a:1, b:1], [a:1, b:2], [a:2, b:1], [a:2, b:2] \rangle \\ R_2 \hat{\bowtie}_{true} R_1 & = & \langle [a:1, b:1], [a:2, b:1], [a:1, b:2], [a:2, b:2] \rangle \end{array}$$

Before introducing the algorithm, let us have a look at the size of the search space. Since the order-preserving join is associative but not commutative, the input to the algorithm must be a sequence of join operators or, likewise, a sequence of relations to be joined. The output is then a fully parenthesized expression. Given a sequence of n binary associative but not commutative operators, the number of fully parenthesized expressions is (see [205])

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n > 1 \end{cases}$$

We have that $P(n) = C(n-1)$, where $C(n)$ are the Catalan numbers defined as $C(n) = \frac{1}{n+1}\binom{2n}{n}$. Since $C(n) = \Omega(\frac{4^n}{n^{3/2}})$, the search space is exponential in size.

The algorithm is inspired by the dynamic programming algorithm for finding optimal parenthesized expressions for matrix-chain multiplication [205]. The differences are that we have to encapsulate the cost function and deal with selections. We give a detailed example application of the algorithm below. This example illustrates (1) the optimization potential, (2) that cross products can be favorable, (3) how to plug in a cost function into the algorithm, and (4) the algorithm itself.

The algorithm itself is broken up into several subroutines. The first is `applicable-predicates` (see Fig. 3.25). Given a sequence of relations $R_i, \ldots, R_j$ and a set of predicates, it retrieves those predicates applicable to the result of the join of the relations. Since joins and selections can be reordered freely, the only condition for a predicate to be applicable is that all its free variables are bound by the given relations.

The second subroutine is the most important and intrigued. It fills several arrays with values in a bottom-up manner. The third subroutine then builds the query evaluation plan using the data in the arrays.

The subroutine `construct-bushy-tree` takes as input a sequence $R_1, \ldots, R_n$ of relations to be joined and a set $\mathcal{P}$ of predicates to be applied. For every possible subsequence $R_i, \ldots, R_j$, the algorithm finds the best plan to join these relations. Therefor,

```
applicable-predicates(R, P)

01  B = ∅
02  foreach p ∈ P
03     IF (F(p) ⊆ A(R))
04        B+ = p
05  return B
```

Figure 3.25: Subroutine `applicable-predicates`

```
construct-bushy-tree(R, P)

01  n = |R|
02  for i = 1 to n
03     B =applicable-predicates(R_i, P)
04     P = P \ B
05     p[i,i] = B
06     s[i,i] = S_0(R_i, B)
07     c[i,i] = C_0(R_i, B)
08  for l = 2 to n
09     for i = 1 to n − l + 1
10        j = i + l − 1
11        B = applicable-predicates(R_{i..j}, P)
12        P = P \ B
13        p[i,j] = B
14        s[i,j] = S_1(s[i,j−1], s[j,j], B)
15        c[i,j] = ∞
16        for k = i to j − 1
17           q = c[i,k] + c[k+1,j] + C_1(s[i,k], s[k+1,j], B)
18           IF (q < c[i,j])
19              c[i,j] = q
20              t[i,j] = k
```

Figure 3.26: Subroutine `construct-bushy-tree`

it determines some k such that the cheapest plan joins the intermediate results for $R_i, \ldots, R_k$ and $R_{k+1}, \ldots, R_j$ by its topmost join. For this it is assumed that for all k the best plans for joining $R_i, \ldots, R_k$ and $R_{k+1}, \ldots, R_j$ are known. Instead of directly storing the best plan, we remember (1) the costs of the best plan for $R_i, \ldots, R_j$ for all $1 \leq i \leq j \leq n$ and (2) the k where the split takes place. More specifically, the array $c[i, j]$ contains the costs of the best plan for joining $R_i, \ldots, R_j$, and the array $t[i, j]$ contains the k such that this best plan joins $R_i, \ldots, R_k$ and $R_{k+1}, \ldots, R_j$ with its topmost join. For every sequence $R_i, \ldots, R_j$, we also remember the set of predicates that can be applied to it, excluding those that have been applied earlier. These applicable predicates are contained in $p[i, j]$. Still, we are not done. All cost functions we know use some kind of statistics on the argument relation(s) in order to compute the costs of some operation. Since we want to be generic with respect to the cost function, we

```
extract-plan(R, t, p)

01   return extract-subplan(R, t, p, 1, |R|)

extract-subplan(R, t, p, i, j)

01   IF (j > i)
02     X = extract-subplan(R, t, p, i, t[i, j])
03     Y = extract-subplan(R, t, p, t[i, j] + 1, j)
04     return X ⋈_{p[i,j]} Y
05   else
06     return σ̂_{p[i,i]}(R_i)
```

Figure 3.27: Subroutine `extract-plan` and its subroutine

encapsulate the computation of statistics and costs within functions S_0, C_0, S_1, and C_1. The function S_0 retrieves statistics for base relations. The function C_0 computes the costs of retrieving (part of) a base relation. Both functions take a set of applicable predicates as an additional argument. The function S_1 computes the statistics for intermediate relations. Since the result of joining some relations $R_i, \ldots, R_j$ may occur in many different plans, we compute it only once and store it in the array s. C_1 computes the costs of joining two relations and applying a set of predicates. Below, we show how concrete (simple) cost and statistics functions can look like.

Given the above, the algorithm (see Fig. 3.26) fills the arrays in a bottom-up manner by first computing for every base relation the applicable predicates, the statistics of the result of applying the predicates to the base relation and the costs for computing these intermediate results, i.e. for retrieving the relevant part of the base relation and applying the predicates (lines 02-07). Note that this is not really trivial if there are several index structures that can be applied. Then computing C_0 involves considering different access paths. Since this is an issue orthogonal to join ordering, we do not detail on it.

After we have the costs and statistics for sequences of length one, we compute the same information for sequences of length two, three, and so on until n (loop starting at line 08). For every length, we iterate over all subsequences of that length (loop starting at line 09). We compute the applicable predicates and the statistics. In order to determine the minimal costs, we have to consider every possible split point. This is done by iterating the split point k from i to $j - 1$ (line 16). For every k, we compute the cost and remember the k that resulted in the lowest costs (lines 17-20).

The last subroutine takes the relations, the split points (t), and the applicable predicates (p) as its input and extracts the plan. The whole plan is extracted by calling `extract-plan`. This is done by instructing `extract-subplan` to retrieve the plan for all relations. This subroutine first determines whether the plan for a base relation or that of an intermediate result is to be constructed. In both cases, we did a little cheating here to keep things simple. The plan we construct for base relations does not take the above-mentioned index structures into account but simply applies a selection to a base relation instead. Obviously, this can easily be corrected. We also give the join operator the whole set of predicates that can be applied. That is, we do

not distinguish between join predicates and other predicates that are better suited for a selection subsequently applied to a join. Again, this can easily be corrected.

Let us have a quick look at the complexity of the algorithm. Given n relations with m attributes in total and p predicates, we can implement `applicable-predicates` in $O(pm)$ by using a bit vector representation for attributes and free variables and computing the attributes for each sequence $R_i, \ldots, R_j$ once upfront. The latter takes $O(n^2 m)$.

The complexity of the routine `construct-bushy-tree` is determined by the three nested loops. We assume that S_1 and C_1 can be computed in $O(p)$, which is quite reasonable. Then, we have $O(n^3 p)$ for the innermost loop, $O(n^2)$ calls to `applicable-predicates`, which amounts to $O(n^2 pm)$, and $O(n^2 p)$ for calls of S_1. Extracting the plan is linear in n. Hence, the total runtime of the algorithm is $O(n^2(n+m)p)$

In order to illustrate the algorithm, we need to fix the functions S_0, S_1, C_0 and C_1. We use the simple cost function C_{out}. As a consequence, the array s simply stores cardinalities, and S_0 has to extract the cardinality of a given base relation and multiply it by the selectivities of the applicable predicates. S_1 multiplies the input cardinalities with the selectivities of the applicable predicates. We set C_0 to zero and C_1 to S_1. The former is justified by the fact that every relation must be accessed exactly once and hence, the access costs are equal in all plans. Summarizing, we define

$$\begin{aligned} S_0(R, \mathcal{B}) &:= |R| \prod_{p \in \mathcal{B}} f(p) \\ S_1(x, y, \mathcal{B}) &:= xy \prod_{p \in \mathcal{B}} f(p) \\ C_0(R, \mathcal{B}) &:= 0 \\ C_1(x, y, \mathcal{B}) &:= S_1(x, y, \mathcal{B}) \end{aligned}$$

where $\mathcal{B}$ is a set of applicable predicates and for a single predicate p, $f(p)$ returns its selectivity.

We illustrate the algorithm by an example consisting of four relations $R_1, \ldots, R_4$ with cardinalities $|R_1| = 200$, $|R_2| = 1$, $|R_3| = 1$, $|R_4| = 20$. Besides, we have three predicates $p_{i,j}$ with $\mathcal{F}(p_{i,j}) \subseteq \mathcal{A}(R_i) \cup \mathcal{A}(R_j)$. They are $p_{1,2}$, $p_{3,4}$, and $p_{1,4}$ with selectivities 1/2, 1/10, 1/5.

Let us first consider an example plan and its costs. The plan

$$((R_1 \bowtie_{p_{1,2}} R_2) \bowtie_{true} R_3) \bowtie_{p_{1,4} \wedge p_{3,4}} R_4$$

has the costs $240 = 100 + 100 + 40$.

For our simple cost function, the algorithm `construct-bushy-tree` will fill the array s with the initial values:

s			
200			
	1		
		1	
			20

After initilization, the array c has 0 everywhere in its diagonal and the array p empty sets.

For $l = 2$, the algorithm produces the following values:

l	i	j	k	s[i,j]	q	current c[i,j]	current t[i,j]
2	1	2	1	100	100	100	1
2	2	3	2	1	1	1	2
2	3	4	3	2	2	2	3

For $l = 3$, the algorithm produces the following values:

l	i	j	k	s[i,j]	q	current c[i,j]	current t[i,j]
3	1	3	1	200	101	101	1
3	1	3	2	200	200	101	1
3	2	4	2	2	4	4	2
3	2	4	3	2	3	3	3

For $l = 4$, the algorithm produces the following values:

l	i	j	k	s[1,4]	q	current c[1,4]	current t[1,4]
4	1	4	1	40	43	43	1
4	1	4	2	40	142	43	1
4	1	4	3	40	141	43	1

where for each k the value of q (in the following table denoted by q_k) is determined as follows:

$$\begin{array}{rcrcrcrcrcr} q_1 & = & c[1,1] & + & c[2,4] & + & 40 & = & 0+3+40 & = & 43 \\ q_2 & = & c[1,2] & + & c[3,4] & + & 40 & = & 100+2+40 & = & 142 \\ q_3 & = & c[1,3] & + & c[4,4] & + & 40 & = & 101+0+40 & = & 141 \end{array}$$

Collecting all the above $t[i,j]$ values leaves us with the following array as input for `extract-plan`:

$i \setminus j$	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

The function `extract-plan` merely calls `extract-subplan`. For the latter, we give the call hierarchy and the result produced:

```
000 extract-plan(..., 1, 4)
100   extract-plan(..., 1, 1)
200   extract-plan(..., 2, 4)
210     extract-plan(..., 2, 3)
211       extract-plan(..., 2, 2)
212       extract-plan(..., 3, 3)
210     return (R2 ⋈true R3)
220     extract-plan(..., 4, 4)
200   return ((R2 ⋈true R3) ⋈p3,4 R4)
000 return (R1 ⋈p1,2∧p1,4 ((R2 ⋈true R3) ⋈p3,4 R4))
```

The total cost of this plan is $c[1,4] = 43$.

3.6 Characterizing Search Spaces

3.6.1 Complexity Thresholds

The complexity results presented in Section 3.1.6 show that most classes of join ordering problems are NP-hard. However, it is quite clear that some instances of the join ordering problem are simpler than others. For example, consider a query graph which is a clique in n relations $R_1, \ldots, R_n$. Further assume that each R_i has cardinality 2^i and all join selectivities are $1/2$ (i.e. $f_{i,j} = 1/2$ for all $1 \leq i, j \leq n, i \neq j$). Obviously, this problem is easy to optimize although the query graph is clique. In this section we present some ideas on how the complexity of instances of the join ordering problem is influenced by certain parameters.

How can we judge the complexity of a single instance of a join ordering problem? Using standard complexity theory, for single problem instances we easily derive an algorithm that works in $\Theta(1)$. Hence, we must define other complexity measures. Consider our introductory join ordering problem. A simple greedy algorithm that orders relations according to their cardinality produces an optimal solution for it. Hence, one possibility to define the problem complexity would be how far a solution produced by typical heuristics for join ordering differ from the optimal solution. Another possibility is to use randomized algorithms like iterative improvement of simulated annealing and see how far the plans generated by them deviate from the optimal plan. These approaches have the problem that the results may depend on the chosen algorithm. This can be avoided by using the following approach. For each join ordering problem instance, we compute the fraction of good plans compared to all plans. Therefor, we need a measure of "good". Typical examples thereof would be to say a plan is "good" if it does not deviate more than 10% or a factor of two from the optimal plan.

If these investigations were readily available, there are certain obvious benefits [497]:

1. The designer of an optimizer can classify queries such that heuristics are applied where they guarantee success; cases where they are bound to fail can be avoided. Furthermore, taking into account the vastly different run time of the different join ordering heuristics and probabilistic optimization procedures, the designer of an optimizer can choose the method that achieves a satisfactory result with the least effort.

2. The developer of search procedures and heuristics can use this knowledge to design methods solving hard problems (as exemplified for graph coloring problems [420]).

3. The investigator of different join ordering techniques is able to (1) consciously design challenging benchmarks and (2) evaluate existing benchmarks according to their degree of challenge.

The kind of investigation presented in this section first started in the context of artificial intelligence where a paper by Cheeseman, Kanefsky, and Taylor [151] spurred a whole new branch of research where the measures to judge the complexity of problem instances was investigated for many different NP-complete problems like satisfiabil-

ity [151, 208, 320, 592], graph coloring [151], Hamiltonian circuits [151], traveling salesman [151], and constraint satisfaction [892].

We only present a small fraction of all possible investigations. The restrictions are that we do not consider all parameters that possibly influence the problem complexity, we only consider left-deep trees, and we restrict ourselves to the cost function C_{hj}. The join graphs are randomly generated. Starting with a circle, we randomly added edges until a clique is reached. The reader is advised to carry out his or her own experiments. Therefor, the following pointer into the literature might be useful. Lanzelotte and Valduriez provide an object-oriented design for search strategies [513]. This allows easy modification and even the exchange of the plan generator's search strategy.

Search Space Analysis

The goal of this section is to determine the influence of the parameters on the search space of left-deep join trees. More specifically, we are interested in how a variation of the parameters changes the percentage of good solutions among all solutions. The quality of a solution is measured by the factor its cost deviates from the optimal permutation. For this, all permutations have to be generated and evaluated. The results of this experiment are shown in Figures 3.28 and 3.29. Each single curve accumulates the percentage of all permutations deviating less than a certain factor (given as the label) from the optimum. The accumulated percentages are given at the y-axes, the connectivity at the x-axes. The connectivity is given by the number of edges in the join graph. The curves within the figures are organized as follows. Figure 3.28 (3.29) shows varying mean selectivity values (relation sizes) and variances where the mean selectivity values (relation sizes) increase from top to bottom and the variances increase from left to right.

Note that the more curves are visible and the lower their y-values, the harder is the problem. We observe the following:

- all curves exhibit a minimum value at a certain connectivity
- which moves with increasing mean values to the right;
- increasing variances does not have an impact on the *minimum connectivity*,
- problems become less difficult with increasing mean values.

These findings can be explained as follows. With increasing connectivity, the join ordering problem becomes more complex up to a certain point and then less complex again. To see this, consider the following special though illustrative case. Assume an almost equal distribution of the costs of all alternatives between the worst case and optimal costs, equal relation sizes, and equal selectivities. Then the optimization potential *worst case/optimum* is 1 for connectivity 0 and cliques. In between, there exists a connectivity exhibiting the maximum optimization potential. This connectivity corresponds to the minimum connectivity of Figures 3.28 and 3.29.

There is another factor which influences the complexity of a single problem instance. Consider joining n relations. The problem becomes less complex if after joining $i < n$ relations the intermediate result becomes so small that the accumulated costs of the subsequent $n - i$ joins are small compared to the costs of joining the first i relations. Hence, the ordering of the remaining $n - i$ relations does not have a big

influence on the total costs. This is the case for very small relations, small selectivities, or high connectivities. The greater selectivities and relation sizes are, the more relations have to be joined to reach this critical size of the intermediate result. If the connectivity is enlarged, this critical size is reached earlier. Since the number of selectivities involved in the first few joins is small regardless of the connectivity, there is a lower limit to the number of joined relations required to arrive at the critical intermediate result size. If the connectivity is larger, this point is reached earlier, but there exists a lower limit on the connectivity where this point is reached. The reason for this lower limit is that the number of selectivities involved in the joins remains small for the first couple of relations, independent of their connectivity. These lines of argument explain subsequent findings, too.

The reader should be aware of the fact that the number of relations joined is quite small (10) in our experiments. Further, as observed by several researchers, if the number of joins increases, the number of "good" plans decreases [286, 827]. That is, increasing the number of relations makes the join ordering problem more difficult.

Heuristics

For analyzing the influence of the parameters on the performance of heuristics, we give the figures for four different heuristics. The first two are very simple. The *minSel* heuristic selects those relations first of which incident join edges exhibit the minimal selectivity. The *recMinRel* heuristic chooses those relations first which result in the smallest intermediate relation.

We also analyzed the two advanced heuristics *IKKBZ* and *RDC*. The *IKKBZ* heuristic [503] is based on an optimal join ordering procedure [428, 503] which is applied to the minimal spanning tree of the join graph where the edges are labeled by the selectivities. The family of *RDC* heuristics is based on the relational difference calculus as developed in [408]. Since our goal is not to benchmark different heuristics in order to determine the best one, we have chosen the simplest variant of the family of *RDC* based heuristics. Here, the relations are ordered according to a certain weight whose actual computation is—for the purpose of this section—of no interest. The results of the experiments are presented in Figure 3.30.

On a first glance, these figures look less regular than those presented so far. This might be due to the non-stable behavior of the heuristics. Nevertheless, we can extract the following observations. Many curves exhibit a peak at a certain connectivity. Here, the heuristics perform worst. The peak connectivity is dependent on the selectivity size but not as regular as in the previous curves. Further, higher selectivities flatten the curves, that is, heuristics perform better at higher selectivities.

Probabilistic Optimization Procedures

Figure 3.31 shows four pictures corresponding to simulated annealing (SA), iterative improvement (II), iterative improvement applied to the outcome of the IKKBZ heuristic (IKKBZ/II) and the RDC heuristic (RDC/II) [408]. The patterns shown in Figure 3.31 are very regular. All curves exhibit a peak at a certain connectivity. The peak connectivities typically coincide with the minimum connectivity of the search space analysis. Higher selectivities result in flatter curves; the probabilistic procedures

perform better. These findings are absolutely coherent with the search space analysis. This is not surprising, since the probabilistic procedures investigate systematically —although with some random influence— a certain part of the search space.

Given a join ordering problem, we can describe its potential search space as a graph. The set of nodes consists of the set of join trees. For every two join trees a and b, we add an edge (a, b) if b can be reached from a by one of the transformation rules used in the probabilistic procedure. Further, with every node we can associate the cost its corresponding join tree.

Having in mind that the probabilistic algorithms are always in danger of being stuck in a local minima, the following two properties of the search space are of interest:

1. the cost distribution of local minima, and

2. the connection cost of low local minima.

Of course, if all local minima are of about the same cost, we do not have to worry, otherwise we do. It would be very interesting to know the percentage of local minima that are close to the global minima.

Concerning the second property, we first have to define the connection cost. Let a and b be two nodes and P be the set of all paths from a to b. The *connection cost* of a and b is then defined as $\min_{p \in P} \max_{s \in p} \{cost(s) | s \neq a, s \neq b\}$. Now, if the connection costs are high, we know that if we have to travel from one local minima to another, there is at least one node we have to pass which has high costs. Obviously, this is bad for our probabilistic procedures. Ioannidis and Kang [439] call a search graph that is favorable with respect to the two properties a *well*. Unfortunately, investigating these two properties of real search spaces is rather difficult. However, Ioannidis and Kang, later supported by Zhang, succeeded in characterizing cost wells in random graphs [439, 440]. They also conclude that the search space comprising bushy trees is better w.r.t. our two properties than the one for left-deep trees.

3.7 Discussion

Choose one of dynamic programming, memoization, permutations as the core of your plan generation algorithm and extend it with the rest of book. ToDo

3.8 Bibliography

ToDo: Oezsu, Meechan [637, 638]

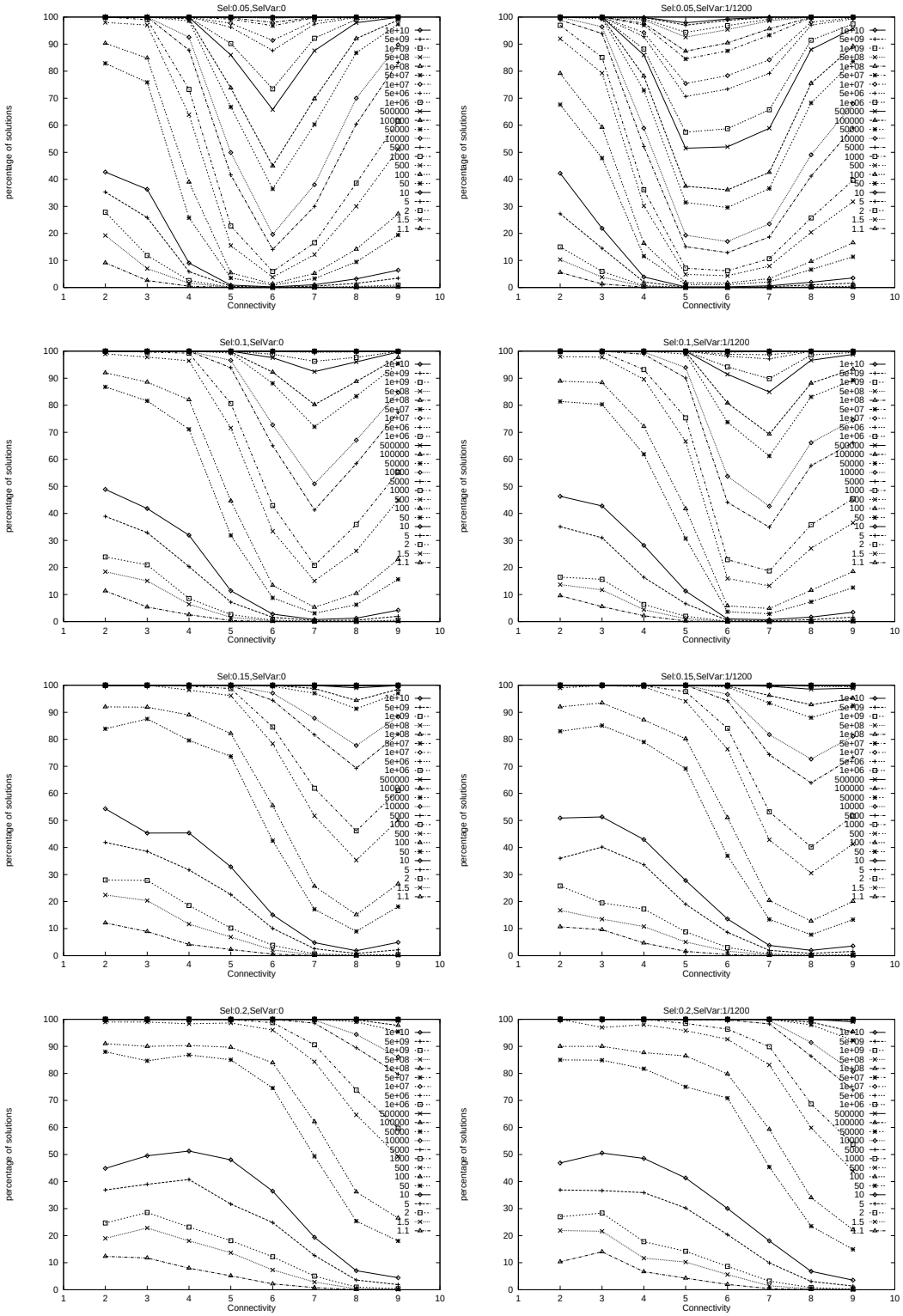


Figure 3.28: Impact of selectivity on the search space

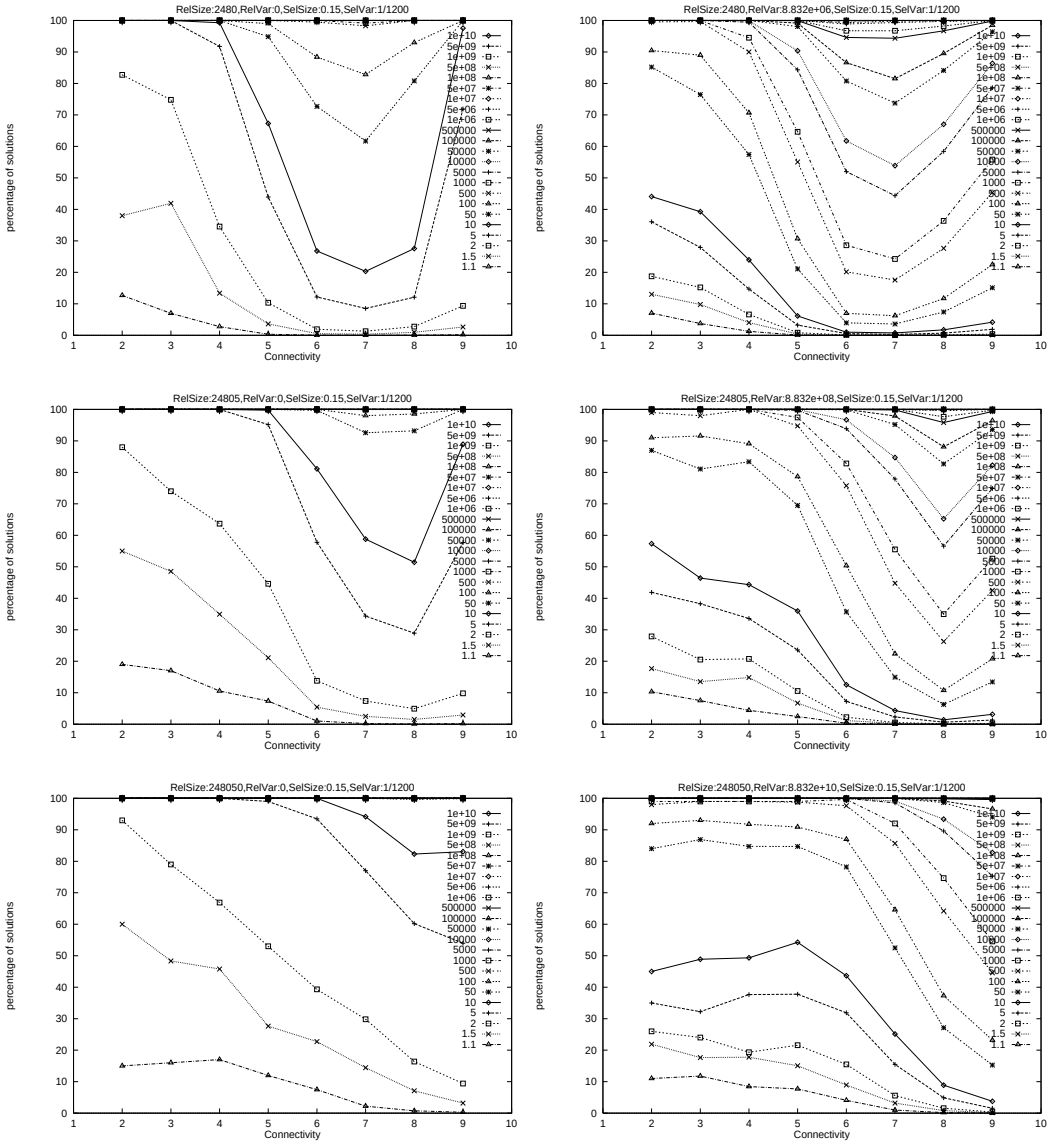


Figure 3.29: Impact of relation sizes on the search space

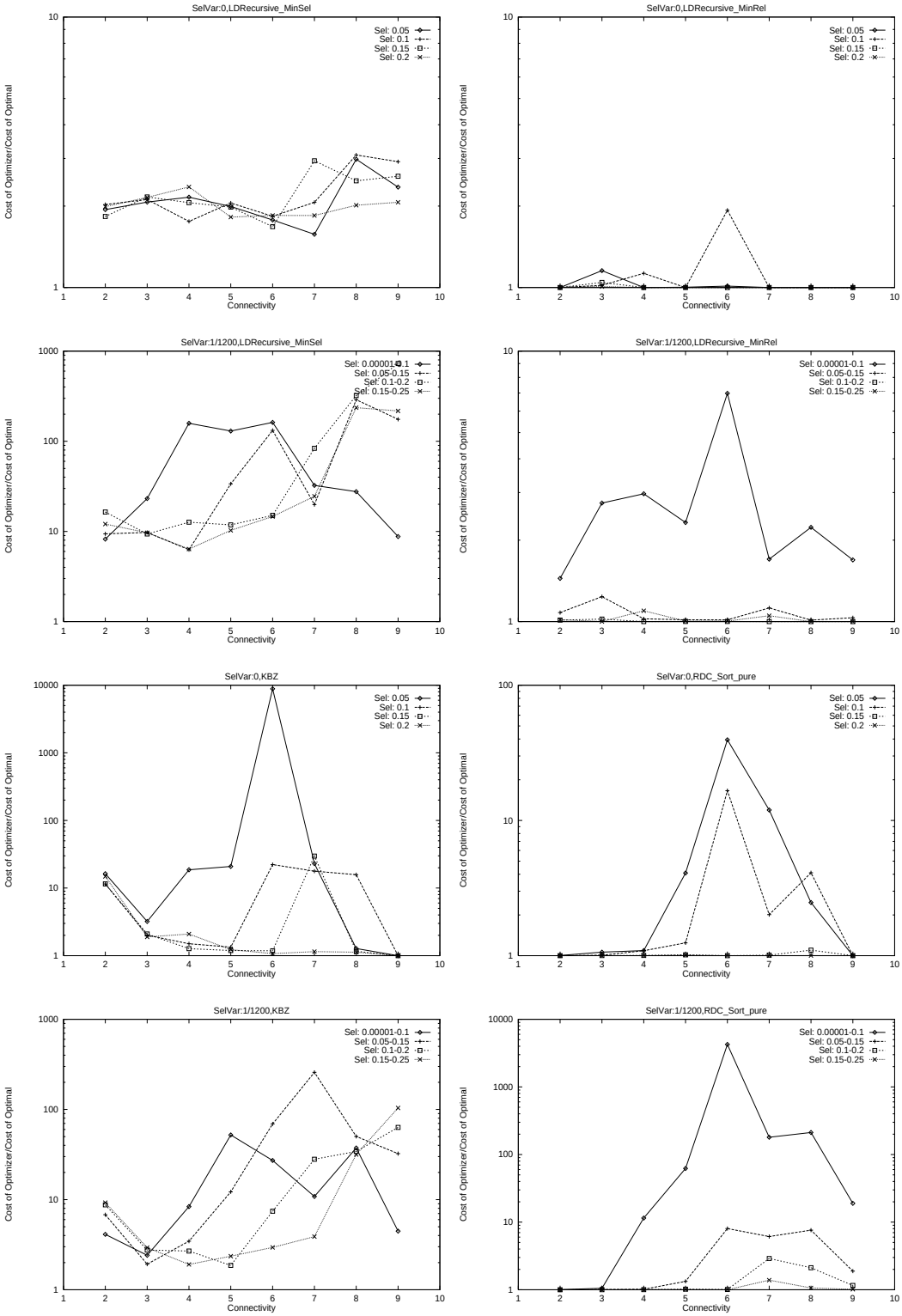


Figure 3.30: Impact of parameters on the performance of heuristics

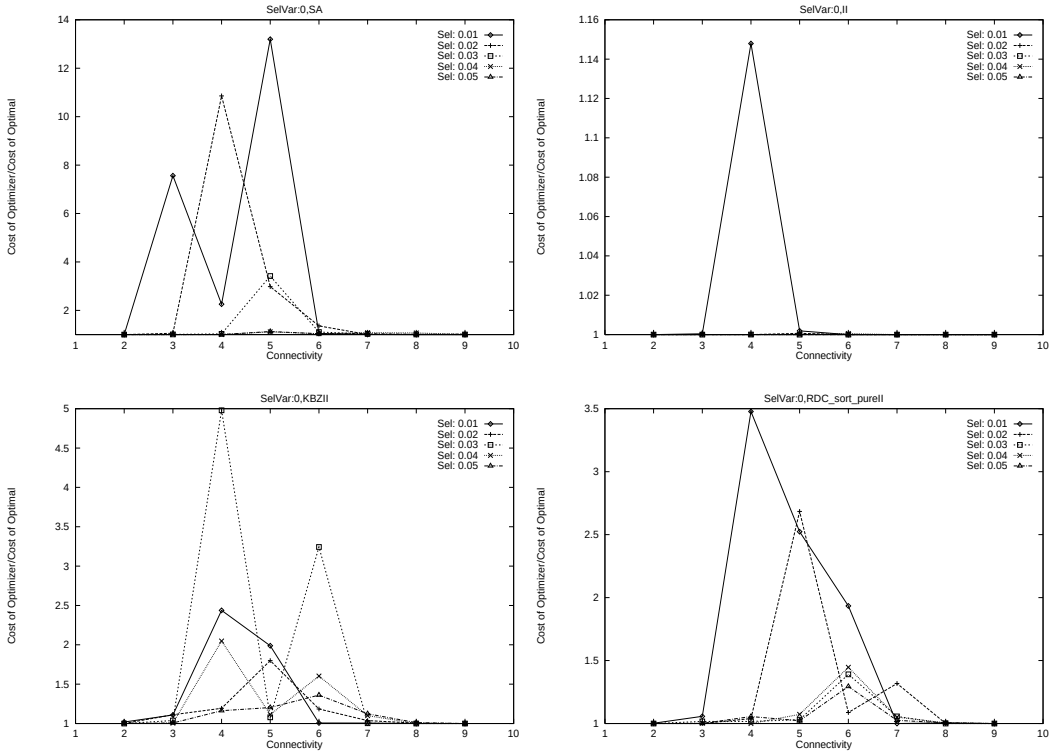


Figure 3.31: Impact of selectivities on probabilistic procedures

Chapter 4

Database Items, Building Blocks, and Access Paths

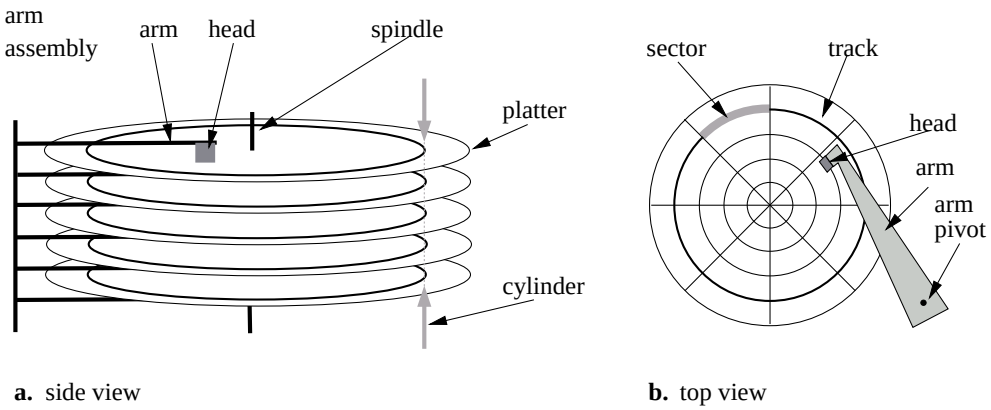
In this chapter we go down to the storage layer and discuss leaf nodes of query execution plans and plan fragments. We briefly recap some notions, but reading a book on database implementation might be helpful [397, 300]. Although alternative storage technologies exist and are being developed [731], databases are mostly stored on disks. Thus, we start out by introducing a simple disk model to capture I/O costs. Then, we say some words about database buffers, physical data organization, slotted pages and tuple identifiers (TIDs), physical record layout, physical algebra, and the iterator concept. These are the basic notions in order to start with the main purpose of this section: giving an overview over the possibilities available to structure the low level parts of a physical query evaluation plan. In order to calculate the I/O costs of these plan fragments, a more sophisticated cost model for several kinds of disk accesses is introduced.

4.1 Disk Drive

Figure 4.1 shows a top and a side view of a typical disk. A disk consists of several platters that rotate around the spindle at a fixed speed. The platters are coated with a magnetic material on at least one of their surfaces. All coated sides are organized into the same pattern of concentric circles. One concentric circle is called a track. All the tracks residing exactly underneath and above each other form a cylinder. We assume that there is only one read/write head for every coated surface.[1] All tracks of a cylinder can be accessed with only minor adjustments at the same time by their respective heads. By moving the arm around the arm pivot, other cylinders can be accessed. Each track is partitioned into sectors. Sectors have a disk specific (almost) fixed capacity of 512 B. The read and write granularity is a sector. Read and write accesses take place while the sector passes under the head.

The top view of Figure 4.1 shows that the outer sectors are longer than the inner sectors. The highest density (e.g. in bits per centimeter) at which bits can be separated is fixed for a given disk. For storing 512 B, this results in a minimum sector length

[1] This assumption is valid for most but not all disks.



a. side view **b.** top view

Figure 4.1: Disk drive assembly

which is used for the tracks of the innermost cylinder. Thus, since sectors on outer tracks are longer, storage capacity is wasted there. To overcome this problem, disks have a varying number of sectors per track. (This is where the picture lies.) Therefore, the cylinders are organized into zones. Every zone contains a fixed number of consecutive cylinders, each having a fixed number of sectors per track. Between zones, the number of sectors per track varies. Outer zones have more sectors per track than inner zones. Since the platters rotate with a fixed angular speed, sectors of outer cylinders can be read faster than sectors of inner cylinders. As a consequence, the throughput for reading and writing outer cylinders is higher than for inner cylinders.

Assume that we sequentially read all the sectors of all tracks of some consecutive cylinders. After reading all sectors of some track, we must proceed to the next track. If it is contained in the same cylinder, then we must (simply) use another head: a *head switch* occurs. Due to calibration, this takes some time. Thus, if all sectors start at the same angular position, we come too late to read the first sector of the next track and have to wait. To avoid this, the angular start positions of the sectors of tracks in the same cylinder are skewed such that this *track skew* compensates for the head switch time. If the next track is contained in another cylinder, the heads have to switch to the next cylinder. Again, this takes time and we miss the first sector if all sectors of a surface start at the same angular positions. *Cylinder skew* is used such that the time needed for this switch does not make us miss to start reading the next sector. In general, skewing works in only one direction.

A sector can be addressed by a triple containing its cylinder, head (surface), and sector number. This triple is called the physical address of a sector. However, disks are accessed using logical addresses. These are called *logical block numbers* (LBN) and are consecutive numbers starting with zero. The disk internally maps LBNs to physical addresses. This mapping is captured in the following table:

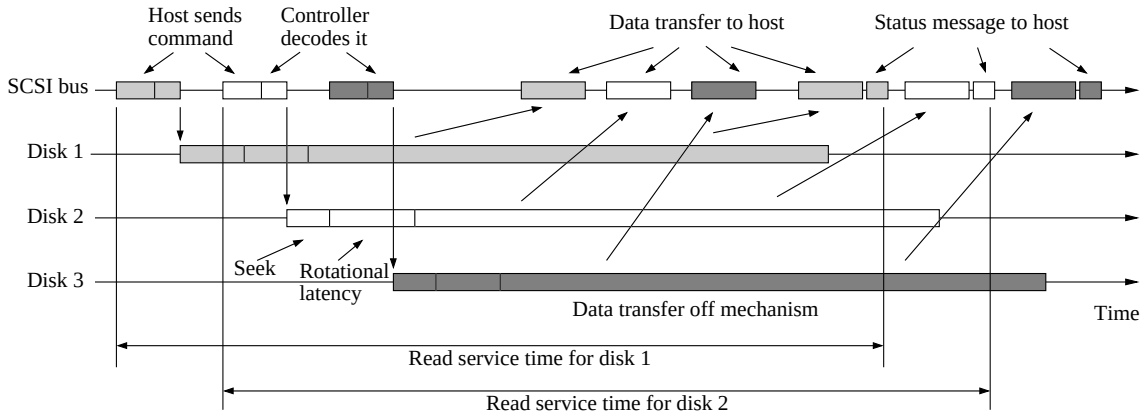


Figure 4.2: Disk drive read request processing

cylinder	track	LBN	number of sectors per track
0	0	0	573
	1	573	573
...	...	...	...
	5	2865	573
1	0	3438	573
...	...	...	...
15041	0	35841845	253
...	...	...	...

However, this ideal view is disturbed by the phenomenon of bad blocks. A *bad block* is one with a defect and it cannot be read or written. After a block with a certain LBN is detected to be bad, it is assigned to another sector. The above mapping changes. In order to be able redirect LBNs, extra space on the disk must exist. Hence, some cylinders, tracks, and sectors are reserved for this purpose. They may be scattered all over the platters. Redirected blocks cause hiccups during sequential read.

Building (see e.g. [620]) and modeling (see e.g. [573, 718, 780, 781, 845, 890]) disk drives is challenging. Whereas the former is not really important when building query compiler, the latter is, as we have to attach costs to query evaluation plans. These costs reflect the amount of time we occupy the resource disk. Since disks are relatively slow, they may become the bottleneck of a database server. Modeling and minimizing disk access (time) is thus an important topic. Consider the case where we want to read a block from a SCSI disk. Simplified, the following actions take place and take their time (see also Fig. 4.2):

1. The host sends the SCSI command.
2. The disk controller decodes the command and calculates the physical address.
3. During the seek the disk drive's arm is positioned such that the according head is correctly placed over the cylinder where the requested block resides. This step consists of several phases.
   (a) The disk controller accelerates the arm.

(b) For long seeks, the arm moves with maximum velocity (coast).

(c) The disk controller slows down the arm.

(d) The disk arm settles for the desired location. The settle times differ for read and write requests. For reads, an aggressive strategy is used. If, after all, it turns out that the block could not be read correctly, we can just discard it. For writing, a more conservative strategy is in order.

4. The disk has to wait until the sector where the requested block resides comes under the head (rotation latency).

5. The disk reads the sector and transfers data to the host.

6. Finally, it sends a status message.

Note that the transfers for different read requests are interleaved. This is possible since the capacity of the SCSI bus is higher than the read throughput of the disk. Also note that we did not mention the operating system delay and congestions on the SCSI bus.

Disk drives apply several strategies to accelerate the above-mentioned round-trip time and access patterns like sequential read. Among them are caching, read-ahead, and command queuing. (discuss interleaving?)

ToDo

The seek and rotation latency times highly depend on the head's position on the platter surface. Let us consider seek time. A good approximation of the seek time where d cylinders have to be travelled is given by

$$\mathrm{s}eektime(d) = \begin{cases} c_1 + c_2\sqrt{d} & d <= c_0 \\ c_3 + c_4 d & d > c_0 \end{cases}$$

where the constants c_i are disk-specific. The constant c_0 indicates the maximum number of cylinders where no coast takes place: seeking over a distance of more than c_0 cylinders results in a phase where the disk arm moves with maximum velocity.

For disk accesses, the database system must be able to estimate the time they take to be executed. First of all, we need the parameters of the disk. It is not too easy to get hold of them, but we can make use of several tools to extract them from a given disk [236, 295, 834, 738, 902, 903]. However, then we have a big problem: when calculating I/O costs, the query compiler has no idea where the head will be when the query evaluation plan emits a certain read (or write) command. Thus, we have to find another solution. In the following, we will discuss a rather simplistic cost model that will serve us to get a feeling for disk behavior. Later, we develop a more realistic model (Section 4.17).

The solution is rather trivial: we sum up all command sending and interpreting times as well the times for positioning (seek and rotation latency) which form by far the major part. Let us call the result *latency time*. Then, we assume an average latency time. This, of course, may result in large errors for a single request. However, on average, the error can be as "low" as 35% [718]. The next parameter is the *sustained read rate*. The disk is assumed to be able to deliver a certain amount of bytes per second while reading data stored consecutively. Of course, considering multi-zone disks, we know that this is oversimplified, but we are still in our simplistic model. Analogously, we have a sustained write rate. For simplicity, we will assume that this is the same

as the sustained read rate. Last, the capacity is of some interest. A hypothetical disk (inspired by disks available in 2004) then has the following parameters:

Model 2004		
Parameter	Value	Abbreviated Name
capacity	180 GB	$\mathtt{D}_{\mathtt{cap}}$
average latency time	5 ms	$\mathtt{D}_{\mathtt{lat}}$
sustained read rate	100 MB/s	$\mathtt{D}_{\mathtt{srr}}$
sustained write rate	100 MB/s	$\mathtt{D}_{\mathtt{swr}}$

The time a disk needs to read and transfer n bytes is then approximated by $\mathtt{D}_{\mathtt{lat}} + n/\mathtt{D}_{\mathtt{srr}}$. Again, this is overly simplistic: (1) due to head switches and cylinder switches, long reads have lower throughput than short reads and (2) multiple zones are not modelled correctly. However, let us use this very simplistic model to get some feeling for disk costs.

Database management system developers distinguish between *sequential* I/O and *random* I/O. For sequential I/O, there is only one positioning at the beginning and then, we can assume that data is read with the sustained read rate. For random I/O, one positioning for every unit of transfer—typically a page of say 8 KB—is assumed. Let us illustrate the effect of positioning by a small example. Assume that we want to read 100 MB of data stored consecutively on a disk. Sequential read takes 5 ms plus 1 s. If we read in blocks of 8 KB where each block requires positioning then reading 100 MB takes 65 s.

Assume that we have a relation of about 100 MB in size, stored on a disk, and we want to read it. Does it take 1 s or 65 s? If the blocks on which it is stored are randomly scattered on disk and we access them in a random order, 65 s is a good approximation. So let us assume that it is stored on consecutive blocks. Assume that we read in chunks of 8 KB. Then,

- other applications,
- other transactions, and
- other read operations of the same query evaluation plan

could move the head away from our reading position. (Congestion on the SCSI bus may also be problem.) Again, we could be left with 65 s. Reading the whole relation with one read request is a possibility but may pose problems to the buffer manager. Fortunately, we can read in chunks much smaller than 100 MB. Consider Figure 4.3. If we read in chunks of 100 8 KB blocks we are already pretty close to one second (within a factor of two).

Note that the interleaving of actions does not necessarily mean a negative impact. This depends on the point of view, i.e. what we want to optimize. If we want to optimize response time for a single query, then obviously the impact of concurrent actions is negative. If, however, we want to optimize resource (here: disk) usage, concurrent actions might help. ToDo?

There are two important things to learn here. First, sequential read is much faster than random read. Second, the runtime system should secure sequential read. The latter point can be generalized: the runtime system of a database management system has, as far as query execution is concerned, two equally important tasks:

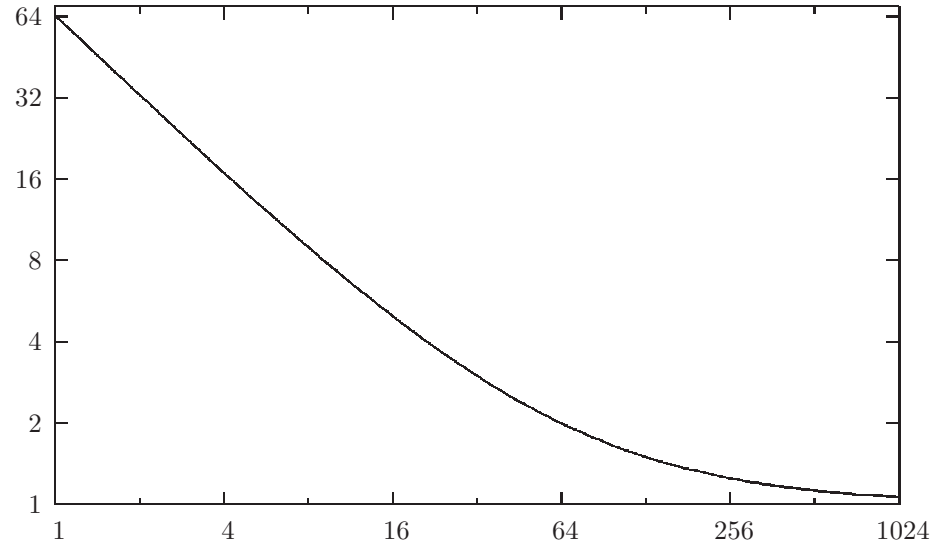


Figure 4.3: Time to read 100 MB from disk (depending on the number of 8 KB blocks read at once)

- allow for efficient query evaluation plans and
- allow for smooth, simple, and robust cost functions.

Typical measures on the database side are

- carefully chosen physical layout on disk
  (e.g. cylinder or track-aligned extents [739, 740, 737], clustering),
- disk scheduling, multi-page requests
  [222, 444, 747, 748, 755, 776, 809, 894, 901],
- (asynchronous) prefetching,
- piggy-back scans,
- buffering (e.g. multiple buffers, replacement strategy from [67] to [583]), and last but not least
- efficient and robust algorithms for algebraic operators [338].

Let us take yet another look at it. 100 MB can be stored on 12800 8 KB pages. Figure 4.4 shows the time to read n random pages. In our simplistic cost model, reading 200 pages randomly costs about the same as reading 100 MB sequentially. That is, reading 1/64th of 100 MB randomly takes as long as reading the 100 MB sequentially. Let us denote by a the positioning time, s the sustained read rate, p the page size, and d some amount of consecutively stored bytes. Let us calculate the

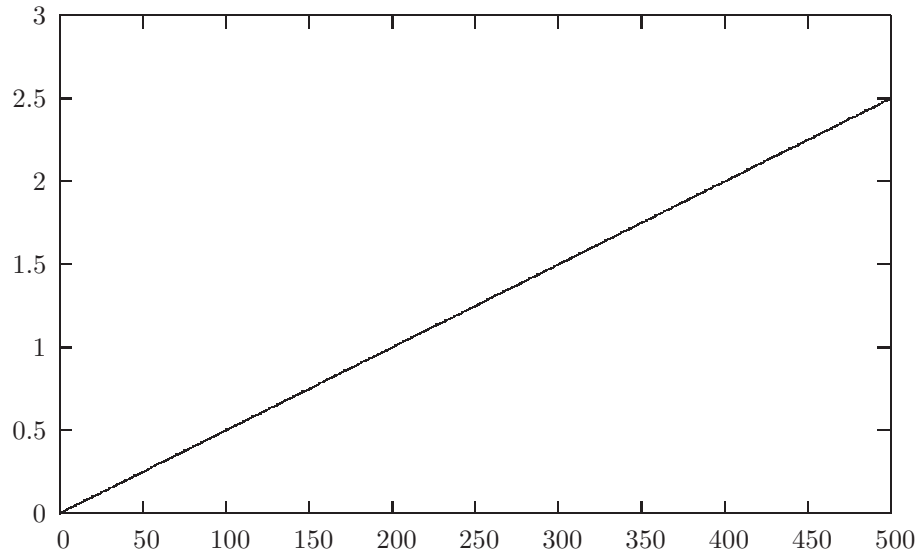


Figure 4.4: Time needed to read n random pages

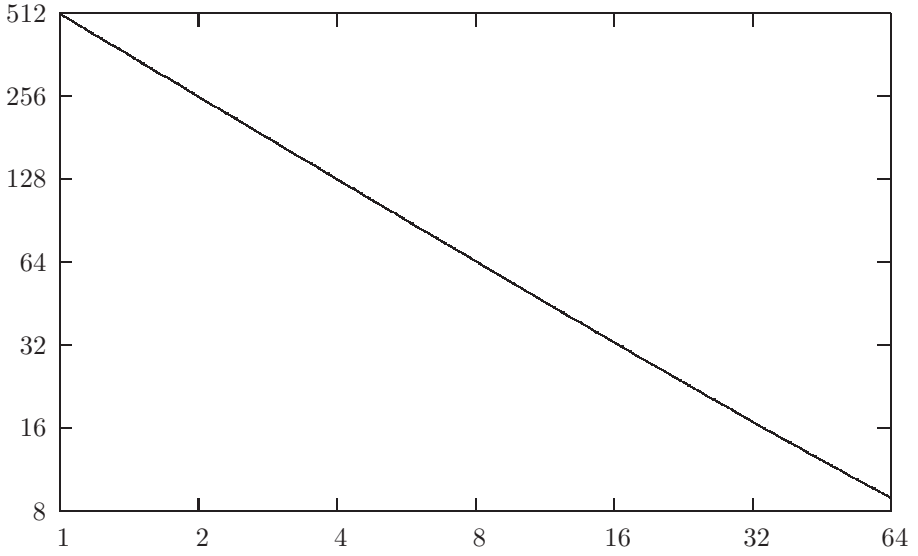


Figure 4.5: Break-even point in fraction of total pages depending on page size

break-even point

$$\begin{aligned} n * (a + p/s) &= a + d/s \\ n &= (a + d/s)/(a + p/s) \\ &= (as + d)/(as + p) \end{aligned}$$

a and s are disk parameters and, hence, fixed. For a fixed d, the break-even point depends on the page size. This is illustrated in Figure 4.5. The x-axis is the page size

p in multiples of 1 K and the y-axis is $(d/p)/n$ for d = 100 MB.

For sequential reads, the page size does not matter. (Be aware that our simplistic model heavily underestimates sequential reads.) For random reads, as long as a single page is read, it matters neither: reading a single page of 1 KB lasts 5.0097656 ms, for an 8 KB page the number is 5.0781250 ms. From all this, we could draw the conclusion that the larger the page the better. However, this is only true for the disk, not, e.g., for the buffer or the SCSI bus. If we need to access only 500 B of a page, then the larger the page the higher the fraction that is wasted. This is not as severe as it sounds. Other queries or transactions might need other parts of the page during a single stay in the buffer. Let us call the fraction of the page that is read by some transaction during a stay in the buffer by utilization. Obviously, the higher the utilization the better is our usage of the main memory in which the buffer resides. For smaller pages, the utilization is typically higher than for larger pages. The frequency by which pages are used is another factor. [358, 359].

Excursion. Consider the root page of a B-tree. It is accessed quite frequently and most of its parts will be used, no matter how large it is. Hence, utilization is always good. Thus, the larger the root page of a B-tree the better. On the other hand, consider a leaf page of a B-tree that is much bigger than main memory. During a single stay of it, only a small fraction of the page will be used. That is, smaller leaf pages are typically better. By converting everything to money instead of time, Gray and Graefe [358] as well as Lomet [554] come to the conclusion that a page size between 8 and 16 KB was a good choice at the end of the last century.

For the less simplistic model of disk access costs developed in Section 4.17, we need to describe a disk drive by a set of parameters. These parameters are summarized in Table 4.1.

Let us close this section by giving upper bounds on seek time and rotational latency. Qyang proved the following theorem which gives a tight upper bound of disk seek time if several cylinders of a consecutive range of cylinders have to be visited [676].

Theorem 4.1.1 (Qyang) *If the disk arm has to travel over a region of C cylinders, it is positioned on the first of the C cylinders and has to stop at $s-1$ of them, then $sD_{seek}(C/s)$ is an upper bound for the seek time.*

The time required for s consecutive sectors in a track of zone i to pass by the head is

$$D_{\text{rot}}(s,i) = sD_{\text{Zscan}}(i) = s\ \frac{D_{\text{rot}}}{D_{\text{Zspt}}(i)} \tag{4.1}$$

A trivial upper bound for the rotational delay is a full rotation.

4.2 Database Buffer

The database buffer

1. is a finite piece of memory,
2. typically supports a limited number of different page sizes (mostly one or two),

D_{cyl}	total number of cylinders
D_{track}	total number of tracks
D_{sector}	total number of sectors
D_{tpc}	number of tracks per cylinder (= number of surfaces)
D_{cmd}	command interpretation time
D_{rot}	time for a full rotation
D_{rdsettle}	time for settle for read
D_{wrsettle}	time for settle for write
D_{hdswitch}	time for head switch
$D_{\text{Z}one}$	total number of zones
$D_{\text{Zcyl}}(i)$	number of cylinders in zone i
$D_{\text{Zspt}}(i)$	number of sectors per track in zone i
$D_{\text{Zspc}}(i)$	number of sectors per cylinder in zone i ($= D_{\text{tpc}} D_{\text{Zspt}}(i)$)
$D_{\text{Zscan}}(i)$	time to scan a sector in zone i ($= D_{\text{rot}} / D_{\text{Zspt}}(i)$)
D_{avgseek}	average seek costs
D_{c_0}	parameter for seek cost function
D_{c_1}	parameter for seek cost function
D_{c_2}	parameter for seek cost function
D_{c_3}	parameter for seek cost function
D_{c_4}	parameter for seek cost function
$D_{\text{seek}}(d)$	cost of a seek of d cylinders
	$D_{\text{seek}}(d) = \begin{cases} D_{c_1} + D_{c_2}\sqrt{d} & \text{if } d \leq D_{c_0} \\ D_{c_3} + D_{c_4} d & \text{if } d > D_{c_0} \end{cases}$
$D_{\text{rot}}(s, i)$	rotation cost for s sectors of zone i ($= s D_{\text{Zscan}}(i)$)

Table 4.1: Disk drive parameters and elementary cost functions

3. is often fragmented into several buffer pools,

4. each having a replacement strategy (typically enhanced by hints).

Given the page identifier, the buffer frame is found by a hashtable lookup. Accesses to the hash table and the buffer frame need to be synchronized. Before accessing a page in the buffer, it must be fixed. These points account for the fact that the costs of accessing a page in the buffer are, therefore, greater than zero.

4.3 Physical Database Organization

We call everything that is stored in the database and relevant for answering queries a *database item*. Let us exclude meta data. In a relational system, a database item can be a relation, a fragment of a relation (if the relation is horizontally or vertically fragmented), a segment, an index, a materialized view, or an index on a materialized view. In object-oriented databases, a database item can be the extent of a class, a named

object, an index and so forth. In XML databases, a database item can be a named document, a collection of documents, or an index. Access operations to database items form the leaves of query evaluation plans.

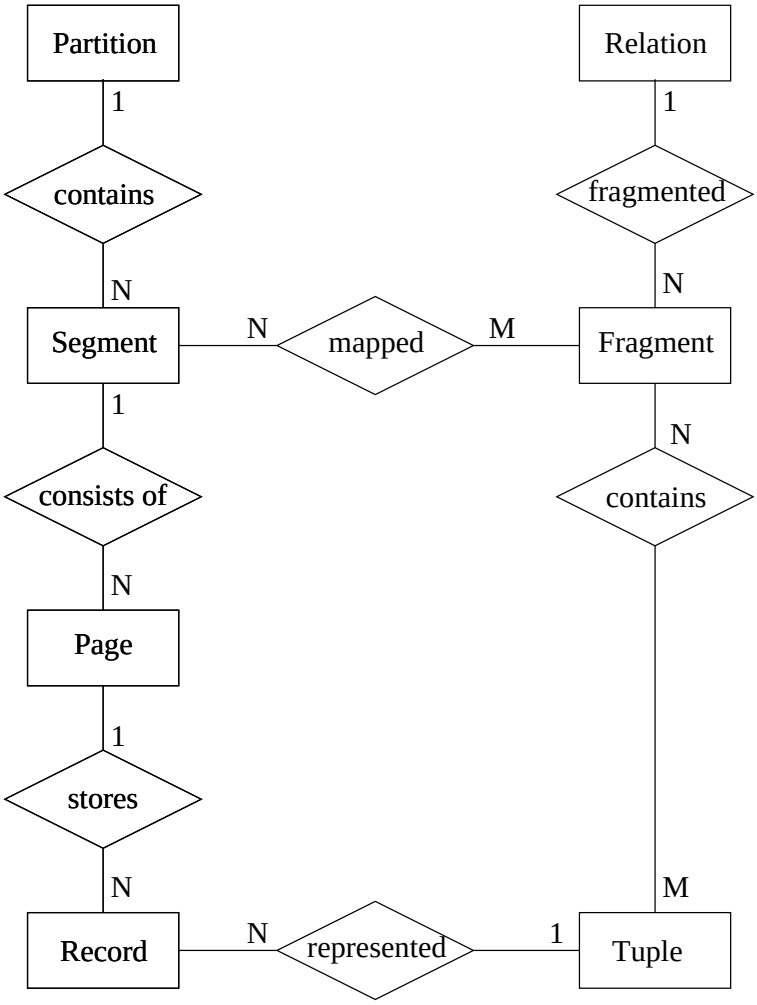


Figure 4.6: Physical organization of a relational database

The physical algebra implemented in the query execution engine of some runtime systems allow to access database items. Since most database items consist of several data items (tuples, objects, documents), these access operations produce a *stream* of data items. This kind of collection-valued access operation is called a *scan*. Consider the simple query

select *
from Student

This query is valid only if the database item (relation) `Student` exists. It could be accessible via a `relation scan` operation `rscan(Student)`. However, in reality we have to consider the physical organization of the database.

Figure 4.6 gives an overview of how relations can be stored in a relational database system. Physical database items can be found on the left-hand side, logical database items on the right-hand side. A fraction of a physical disk is a partition. It can be an operating system file or a raw partition. A partition is organized into several segments. A segment consists of several pages. The pages within a segment are typically accessible by a non-negative integer in $[0, n[$, where n is the number of pages of the segment[2]. Iterative access to all pages of a segment is typically possible. The access is called a *scan*. As there are several types of segments (e.g. data segments, index segments), several kinds of scans exist. Within a page, *physical records* are stored. Each physical record represents a (part of a) tuple of a fragment of a relation.

Fragments are mapped to segments and relations are partitioned into fragments. In the simplest and most common organization, every relation has only one fragment with a one-to-one mapping to segments, and for every tuple there exists exactly one record representing only this tuple. Hence, both of relationships `mapped` and `represented` are one-to-one. However, this organization does not scale well. A relation could be larger than a disk. Even if a large relation, say 180 GB fits on a disk, scanning it takes half an hour (Model 2004). Horizontal partitioning and allocation of the fragments on several disks reduces the scan time by allowing for parallelism. Vertical partitioning is another means of reducing I/O [203]. Here, a tuple is represented by several physical records, each one containing a subset of the tuple's attributes. Since the relationship *mapped* is N:M, tuples from different relations can be stored in the same segment. Furthermore, in distributed database systems some fragments might be stored redundantly at different locations to improve access times [121, 498, 677, 639]. Some systems support clustering of tuples of different relations. For example, department tuples can be clustered with employee tuples such that those employees belonging to the department are close together and close to their department tuple. Such an organization speeds up join processing.

To estimate costs, we need a model of a segment. We assume an extent-based implementation. That is, a segment consists of several extents[3]. Each extent occupies consecutive sectors on disk. For simplicity, we assume that whole cylinders belong to a segment. Then, we can model segments as follows. Each segment consists of a sequence of *extents*. Each extent is stored on *consecutive cylinders*. Cylinders are exclusively assigned to a segment. We then describe each extent j as a pair (F_j, L_j) where F_j is the first and L_j the last cylinder of a consecutive sequence of cylinders. A segment can then be described by a sequence of such pairs. We assume that these pairs are sorted in ascending order. In such a description, an extent may include a zone boundary. Since cost functions are dependent on the zone, we break up cylinder ranges that are not contained in a single zone. The result can be described by a sequence of triples (F_i, L_i, z_i) where F_i and L_i mark a range of consecutive cylinders in a zone z_i. Although the z_i's can be inferred from the cylinder numbers, we include them for clarity. Also of interest are the total number of sectors in a segment and the number of cylinders $S_{\text{cpe}}(i)$ in an extent i. Summarizing, we describe a segment by the parameter given in Table 4.2.

[2]This might not be true. Alternatively, the pages of a partition can be consecutively numbered.

[3]Extents are not shown in Fig. 4.6. They can be included between Partitions and Segments.

S_{ext}	number of extents in the segment
S_{sec}	total number of sectors in the segment ($= \sum_{i=1}^{S_{\text{ext}}} S_{\text{cpe}}(i) D_{\text{Zspc}}(S_{\text{zone}}(i))$)
$S_{\text{first}}(i)$	first cylinder in extent i
$S_{\text{last}}(i)$	last cylinder in extent i
$S_{\text{cpe}}(i)$	number of cylinders in extent i ($= S_{\text{last}}(i) - S_{\text{first}}(i) + 1$)
$S_{\text{zone}}(i)$	zone of extent i

Table 4.2: Segment parameters

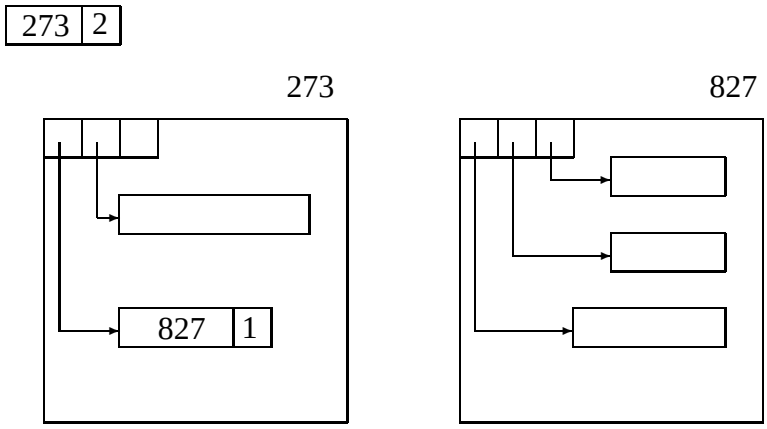


Figure 4.7: Slotted pages and TIDs

4.4 Slotted Page and Tuple Identifier (TID)

Let us briefly review *slotted pages* and the concept of *tuple identifiers (TIDs)* (see Figure 4.7) [40, 39, 555, 816]. Sometimes, *record identifer* or *row identifier* (RID) is used in the literature. A TID consists of (at least) two parts. The first part identifies a page, the second part a slot on a *slotted page*. The slot contains—among other things, e.g. the record's size—a (relative) pointer to the actual record. This way, the record can be moved within the page without invalidating its TID. When a record grows beyond the available space, it is moved to another page and leaves a forward pointer (again consisting of a page and a slot identifier) in its original position. This happened to the TID $[273, 1]$ in Figure 4.7. If the record has to be moved again, the forward pointer is adjusted. This way, at most two page accesses are needed to retrieve a record, given its TID. For evaluating the costs of record accesses, we will assume that the fraction of moved records is known.

4.5 Physical Record Layouts

A physical record represents a tuple, object, or some other logical entity or fraction thereof. In case it represents a tuple, it consists of several fields, each representing the value of an attribute. These values can be integers, floating point numbers, or strings. In case of object-oriented or object-relational systems, the values can also be of a complex type. Tuple identifiers are also possible as attribute values [699]. This

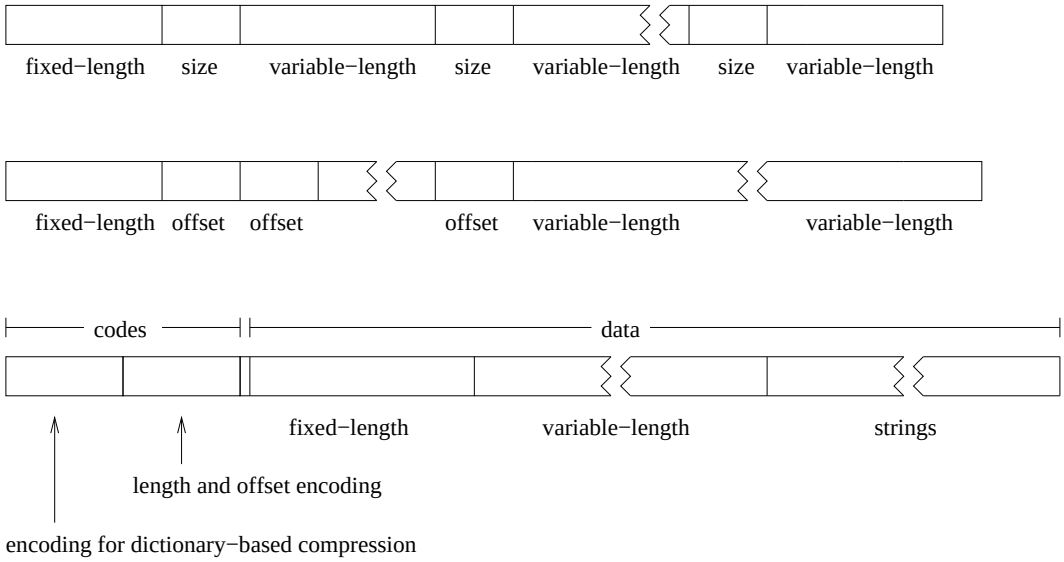


Figure 4.8: Various physical record layouts

can, for example, speed up join processing.

In any case, we can distinguish between types whose values all exhibit the same fixed length and those whose values may vary in length. In a physical record, the values of fixed-length attributes are concatenated and the offset from the beginning of the record to the value of some selected attribute can be inferred from the types of the values preceding it. This differs for values of varying length. Here, several encodings are possible. Some simple ones are depicted in Figure 4.8. The topmost record encodes varying length values as a sequence of pairs of the form $[size, value]$. This encoding has the disadvantage that access to an attribute of varying length is linear in the number of those preceding it. This disadvantage is avoided in the solution presented in the middle. Instead of storing the sizes of the individual values, there is an array containing relative offsets into the physical record. They point to the start of the values. The length of the values can be inferred from these offsets and, in case of the last value, from the total length of the physical record, which is typically stored in its slot. Access to a value of varying size is now simplified to an indirect memory access plus some length calculations. Although this might be cheaper than the first solution, there is still a non-negligible cost associated with an attribute access.

The third physical record layout can be used to represent compressed attribute values and even compressed length information for parts of varying size. Note that if fixed size fields are compressed, their length becomes varying. Access to an attribute now means decompressing length/offset information and decompressing the value itself. The former is quite cheap: it boils down to an indirect memory access with some offset taken from a matrix [885]. The cost of the latter depends on the compression scheme used. It should be clear that accessing an attribute value now is even more expensive. To make the costs of an attribute access explicit was the sole purpose of this small section.

Remark Westmann et al. discuss an efficient implementation of compression and evaluate its performance [885]. Yiannis and Zobel report on experiments with several

compression techniques used to speed up the sort operator. For some of them, the CPU usage is twice as large [**?**].

4.6 Physical Algebra (Iterator Concept)

Physical algebraic operators are mostly implemented as *iterators*. This means that they support the the interface operations `open`, `next`, and `close`. With `open`, the stream of items (e.g. tuples) is initialized. With `next`, the next item on the stream is fetched. When no more items are available, e.g. `next` returns false, `close` can be called to clean up things. The iterator concept is explained in many text books (e.g. [300, 397, 472]) and the query processing survey by Graefe [338]. This basic iterator concept has been extended to better cope with nested evaluation by Westmann in his thesis [883], Westmann et al. [885], and Graefe [342]. The two main issues are separation of storage allocation and initialization, and batched processing. The former splits `open` into resource allocation, initialization of the operator, and initialization of the iterator.

4.7 Simple Scan

Let us come back to the scan operations. A logical operation for scanning relations (which could be called `rscan`) is rarely supported by relational database management systems. Instead, they provide (physical) scans on segments. Since a (data) segment is sometimes called *file*, the correct plan for the above query is often denoted by `fscan(Student)`. Several assumptions must hold: the `Student` relation is not fragmented, it is stored in a single segment, the name of this segment is the same as the relation name, and no tuples from other relations are stored in this segment. Until otherwise stated, we will assume that relations are not partitioned, are stored in a single segment and that the segment can be inferred from the relation's name. Instead of `fscan(Student)`, we could then simply use `Student` to denote leaf nodes in a query execution plan. If we want to use a variable that is bound subsequently to each tuple in a relation, the query

select *
from Student

can be expressed as `Student`$[s]$ instead of `Student`. In this notation, the output stream contains tuples having a single attribute s bound to a tuple. Physically, s will not hold the whole tuple but, for example, a pointer into the buffer where the tuple can be found. An alternative is a pointer to a slot of a slotted page contained in the buffer.

A simple scan is an example for a *building block*. In general, a building block is something that is used as a bottommost operator in a query evaluation plan. Hence, every leaf node of a query evaluation plan is a building block or a part thereof. This is not really a sharp definition, but is sometimes useful to describe the behavior of a query compiler: after their determination, it will leave building blocks untouched even if reorderings are hypothetically possible. Although a building block can be more than a leaf node (scan) of a query evaluation plan, it will never include more than a single

database item. As soon as more database items are involved, we use the notion of *access path*, a term which will become more precise later on when we discuss index usage.

The disk access costs for a simple scan can be derived from the considerations in Section 4.1 and Section 4.17.

4.8 Scan and Attribute Access

Strictly speaking, a plan like $\sigma_{\texttt{age}>30}(\texttt{Student}[s])$ is invalid, since the tuple stream produced by $\texttt{Student}[s]$ contains tuples with a single attribute s. We have a choice. Either we assume that attribute access takes place implicitly, or we make it explicit. Whether this makes sense or not depends on the database management system for which we generate plans. Let us discuss the advantages of explicit attribute retrieval. Assume `s.age` retrieves the age of a student. Then we can write $\sigma_{s.\texttt{age}>30}(\texttt{Student}[s])$, where there is some non-neglectable cost for `s.age`. The expression $\sigma_{\texttt{s.age}>30\wedge\texttt{s.age}<40}(\texttt{Student}[s])$ executes `s.age` twice. This is a bad idea. Instead, we would like to retrieve it once and reuse it later.

This purpose is well-served by the *map* operator (χ). It adds new attributes to a given tuple and is defined as

$$\chi_{a_1:e_1,\ldots,a_n:e_n}(e) := \{t \circ [a_1 : c_1, \ldots, a_n : c_n] | t \in e, c_i = e_i(t) \; \forall \; (1 \leq i \leq n)\}$$

where $\circ$ denotes tuple concatenation and the a_i must not be in $\mathcal{A}(e)$. (Remember that $\mathcal{A}(e)$ is the set of attributes produced by e.) Every input tuple t is extended by new attributes a_i, whose values are computed by evaluating the expression e_i, in which free variables (attributes) are bound to the attributes (variables) provided by t.

The above problem can now be solved by

$$\sigma_{\texttt{age}>30\wedge\texttt{age}<40}(\chi_{\texttt{age}:s.\texttt{age}}(\texttt{Student}[s])).$$

In general, it is beneficial to load attributes as late as possible. The latest point at which all attributes must be read from the page is typically just before a pipeline breaker[4].

To see why this is useful, consider the simple query

select name
from Student
where age > 30

The plan

$$\Pi_n(\chi_{n:s.\texttt{name}}(\sigma_{a>30}(\chi_{a:s.\texttt{age}}(\texttt{Student}[s]))))$$

makes use of this feature, while

$$\Pi_n(\sigma_{a>30}(\chi_{n:s.\texttt{name},a:s.\texttt{age}}(\texttt{Student}[s])))$$

does not. In the first plan the `name` attribute is only accessed for those students with age over 30. Hence, it should be cheaper to evaluate. If the database management system does not support this selective access mechanism, we often find the scan enhanced by a list of attributes that is projected and included in the resulting tuple stream.

[4]The page on which the physical record resides must be fixed until all attributes are loaded. Hence, an earlier point in time might be preferable.

In order to avoid copying attributes from their storage representation to some main memory representation, some database management systems apply another mechanism. They support the evaluation of some predicates directly on the storage representation. These are boolean expressions consisting of simple predicates of the form $A\theta c$ for attributes A, comparison operators θ, and constants c. Instead of a constant, c could also be the value of some attribute or expression thereof given that it can be evaluated before the access to A.

Predicates evaluable on the disk representation are called *SARGable* where *SARG* is an acronym for *search argument*. Note that SARGable predicates may also be good for index lookups. Then they are called *index SARGable*. In case they can not be evaluated by an index, they are called *data SARGable* [750, 832, 313].

Since relation or segment scans can evaluate predicates, we have to extend our notation for scans. Let I be a database item like a relation or segment. Then, $I[v; p]$ scans I, binds each item in I successively to v and returns only those items for which p holds. $I[v; p]$ is equivalent to $\sigma_p(I[v])$, but cheaper to evaluate. If p is a conjunction of predicates, the conjuncts should be ordered such that the attribute access cost reductions described above are reflected (for details see Chapter **??**). Syntactically, we express this by separating the predicates by a comma as in $\texttt{Student}[s; \texttt{age} > 30, \texttt{name like } \text{`}\%\texttt{m}\%\text{'}]$. If we want to make a distinction between SARGable and non-SARGable predicates, we write $I[v; p_s; p_r]$, with p_s being the SARGable predicate and p_r a non-SARGable predicate. Additional extensions like a projection list are also possible.

4.9 Temporal Relations

Scanning a temporal relation or segment also makes sense. Whenever the result of some (partial) query evaluation plan is used more than once, it might be worthwhile to materialize it in some temporary relation. For this purpose, a `tmp` operator evaluates its argument expression and stores the result relation in a temporary segment. Consider the following example query.

select e.name, d.name
from Emp e, Dept d
where e.age $>$ 30 **and** e.age $<$ 40 **and** e.dno = d.dno

It can be evaluated by

$$\texttt{Dept}[d] \bowtie^{\texttt{nl}}_{e.\texttt{dno}=d.\texttt{dno}} \sigma_{e.\texttt{age}>30 \wedge e.\texttt{age}<40}(\texttt{Emp}[d]).$$

Since the inner (right) argument of the nested-loop join is evaluated several times (once for each department), materialization may pay off. The plan then looks like

$$\texttt{Dept}[d] \bowtie^{\texttt{nl}}_{e.\texttt{dno}=d.\texttt{dno}} \text{Tmp}(\sigma_{e.\texttt{age}>30 \wedge e.\texttt{age}<40}(\texttt{Emp}[d])).$$

If we choose to factorize and materialize a common subexpression, the query evaluation plan becomes a DAG. Alternatively, we could write a small "program" that has some statements materializing some expressions which are then used later on. The last expression in a program determines its result. For our example, the program looks as follows.

1. $R_{\texttt{tmp}} = \sigma_{e.\texttt{age}>30 \wedge e.\texttt{age}<40}(\texttt{Emp}[d])$;
2. $\texttt{Dept}[d] \bowtie^{\texttt{nl}}_{e.\texttt{dno}=d.\texttt{dno}} R_{\texttt{tmp}}[e]$

The disk costs of writing and reading temporary relations can be calculated using the considerations of Section 4.1.

4.10 Table Functions

A *table function* is a function that returns a relation [563]. An example is `Primes(int from, int to)`, which returns all primes between `from` and `to`, e.g. via a sieve-method. It can be used in any place where a relation name can occur. The query

select *
from TABLE(Primes(1,100)) as p

returns all primes between 1 and 100. The attribute names of the resulting relation are specified in the declaration of the table function. Let us assume that for `Primes` a single attribute `prime` is specified. Note that table functions may take parameters. This does not pose any problems, as long as we know that `Primes` is a table function and we translate the above query into $\texttt{Primes}(1, 100)[p]$. Although this looks exactly like a table scan, the implementation and cost calculations are different.

Consider the following query where we extract the years in which we expect a special celebration of Anton's birthday.

select *
from Friends f,
TABLE(Primes(
CURRENT_YEAR, EXTRACT(YEAR FROM f.birthday) + 100)) as p
where f.name = 'Anton'

The result of the table function depends on our friend Anton. Hence, a join is no solution. Instead, we have to introduce a new kind of join, the *d-join* where the *d* stands for dependent. It is defined as

$$R < S >= \{t \circ s | t \in T, s \in S(t)\}.$$

The above query can now be evaluted as

$$\chi_{\text{b}:EXTRACT_YEAR(f.birthday)+100}(\sigma_{\text{f}.name='Anton'}(\texttt{Friends}[f])) < \texttt{Primes}(\texttt{c},\texttt{b})[p] >$$

where we assume that some global entity c holds the value of `CURRENT_YEAR`.

Let us do the above query for all friends. We just have to drop the `where` clause. Obviously, this results in many redundant computations of primes. At the SQL level, using the birthday of the youngest friend is beneficial:

select *
from Friends f,
TABLE(Primes(
CURRENT_YEAR, (**select** max(birthday) **from** Friends) + 100)) as p
where p.prime $\geq$ f.birthday

At the algebraic level, this kind of optimizations will be considered in Section **??**.

ToDo?

Things can get even more involved if table functions can consume and produce relations, i.e. arguments and results can be relations.

Little can be said about the disk costs of table functions. They can be zero if the function is implemented such that it does not access any disks (files stored there), but it can also be very expensive if large files are scanned each time it is called. One possibility is to let the database administrator specify the numbers the query optimizer needs. However, since parameters are involved, this is not really an easy task. Another possibility is to measure the table function's behavior whenever it is executed, and learn about its resource consumption.

4.11 Indexes

There exists a plethora of different index structures. In the context of relational database management systems, the most versatile and robust index is the B-tree or variants/improvements thereof (e.g. []). It is implemented in almost every commercial database management system. Some support hash-indexes (e.g. []). Other data models or specific applications need specialized indexes. There exist special index structures for indexing path expressions in object-oriented databases (e.g. []) and XML databases (e.g. []). Special purpose indexes include join indexes (e.g. [395, 854]) multi-dimensional indexes (e.g. []), variant (projection) indexes [625], small materialized aggregates [598], bitmap indexes [], and temporal indexes (e.g. []). We cannot discuss all indexes and their exploitations for efficient query evaluation. This fills more than a single book. Instead, we concentrate on B-tree indexes. In general, a B-tree can be used to index several relations. We only discuss cases where B-trees index a single relation.

The *search key* (or key for short) of an index is the sequence of attributes of the *indexed relation* over which the index is defined. A key is a *simple key* if it consists of a single attribute. Otherwise, it is a *complex key*. Each entry in the B-tree's leaf page consists of pairs containing the key values and a sequence of tuple identifiers (typically sorted by increasing page number). Every tuple with a TID in this list satisfies the condition that its indexed attribute's values are equal to the key values. If for every sequence of key values there is at most one such tuple, we have a *unique* index, otherwise a *non-unique index*.

The leaf entries may contain values from additional (non-key) attributes. Then we call the index *attribute data added* and the additional attributes *data attributes*. If the index contains all attributes of the indexed relation—in its key or data attributes—storing the relation is no longer necessary. The result is an *index-only relation*. In this case, the concept of tuple identifiers is normally no longer used since tuples can now be moved frequently, e.g. due to a leaf page split. This has two consequences. First, the data part does not longer contain the TID. Second, other indexes on the index-only relation cannot have tuple identifiers in their data part either. Instead, they use the key of the index-only relation to uniquely reference a tuple. For this to work, we must have a unique index.

B-trees can be either *clustered* or *non-clustered* indexes. In a clustered index, the tuple identifiers in the list of leaf pages are ordered according to their page numbers.

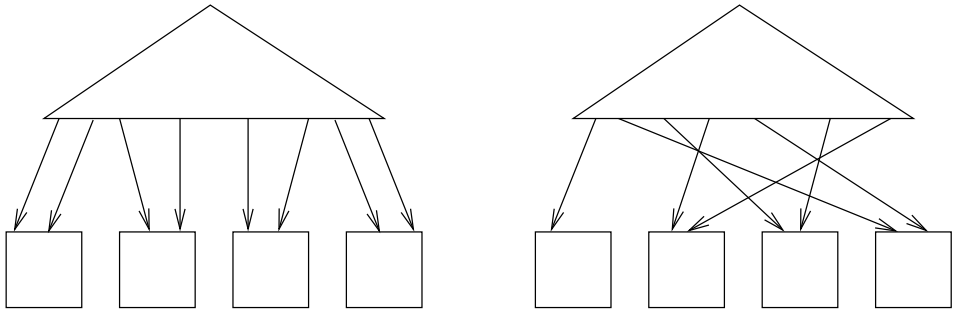


Figure 4.9: Clustered vs. non-clustered index

Otherwise, it is a *non-clustered* index[5]. Figure 4.9 illustrates this. Range queries result in sequential access for clustered indexes and in random access for non-clustered indexes.

4.12 Single Index Access Path

4.12.1 Simple Key, No Data Attributes

Consider the *exact match query*

select name
from Emp
where eno = 1077

If there exists a unique index on the key attribute `eno`, we can first access the index to retrieve the TID of the employee tuple satisfying `eno = 1077`. Another page access yields the tuple itself which constitutes the result of the query. Let $\texttt{Emp}_{\texttt{eno}}$ be the index on `eno`, then we can descend the B-tree, using 1077 as the search key. A predicate that can be used to descend the B-tree or, in general, governing search within an index structure, is called an *index sargable predicate*.

For the example query, the index scan, denoted as $\texttt{Emp}_{\texttt{eno}}[x; \texttt{eno} = 1077]$, retrieves a single leaf node entry with attributes `eno` and `TID`. Similar to the regular scan, we assume x to be a variable holding a pointer to this index entry. We use the notations `x.eno` and `x.TID` to access these attributes. To dereference the TID, we use the map (χ) operator and a dereference function `deref` (or $*$ for short). It turns a TID into a pointer in the buffer area. This of course requires the page to be loaded, if it is not in the buffer yet. The complete plan for the query is

$$\Pi_{\texttt{name}}(\chi_{e:*(x.\text{TID}),\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{eno}}[x; \texttt{eno} = 1077]))$$

where we computed several new attributes with one χ operator. Note that they are dependent on previously computed attributes and, hence, the order of evaluation does matter.

[5]Of course, any degree of clusteredness may occur and has to be taken into account in cost calculations.

We can make the dependency of the map operator more explicit by applying a d-join. Denote by $\Box$ an operator that returns a single empty tuple. Then

$$\Pi_{\texttt{name}}(\texttt{Emp}_{\texttt{eno}}[x; \texttt{eno} = 1077] < \chi_{e:*(x.\text{TID}),\texttt{name:e.name}}(\Box) >)$$

is equivalent to the former plan. Joins and indexes will be discussed in Section 4.14.

A *range query* like

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35

specifies a range for the indexed attribute. It is evaluated by an index scan with *start* and *stop* conditions. In our case, the start condition is `age` $\geq$ `25`, and the stop condition is `age` $\leq$ `35`. The start condition is used to retrieve the first tuple satisfying it by searching within the B-tree. In our case, 25 is used to descend from the root to the leaf page containing the key 25. Then, all records with keys larger than 25 within the page are searched. Since entries in B-tree pages are sorted on key values, this is very efficient. If we are done with the leaf page that contains 25 and the stop key has not been found yet, we proceed to the next leaf page. This is possible since leaf pages of B-trees tend to be chained. Then all records of the next leaf page are scanned and so on until we find the stop key. The complete plan then is

$$\Pi_{\texttt{name}}(\chi_{e:*(x.\text{TID}),\texttt{name:e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35]))$$

If the index on `age` is non-clustered, this plan results in random I/O. We can turn random I/O into sequential I/O by sorting the result of the index scan on its TID attribute before dereferencing it[6]. This results in the following plan:

$$\Pi_{\texttt{name}}(\chi_{e:*(\text{TID}),\texttt{name:e.name}}(\text{Sort}_{\text{TID}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \text{TID}])))$$

Here, we explicitly included the TID attribute of the index into the projection list.

Consider a similar query which demands the output to be sorted:

select name, age
from Emp
where age $\geq$ 25 **and** age $\leq$ 35
order by age

Since an index scan on a B-tree outputs its result ordered on the indexed attribute, the following plan produces the perfect result:

$$\Pi_{\texttt{name},\texttt{age}}(\chi_{e:*(x.\text{TID}),\texttt{name:}e.\texttt{name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35]))$$

On a clustered index this is most probably the best plan. On a non-clustered index, random I/O disturbs the picture. We avoid that by sorting the result of the index scan

[6]This might not be necessary, if `Emp` fits main memory. Then, preferably asynchronous I/O should be used.

on the TID attribute and, after accessing the tuples, restore the order on age as in the following plan:

$$\Pi_{\texttt{name,age}}(\text{Sort}_{\texttt{age}}(\chi_{e:*(\text{TID}),\texttt{name}:e.\texttt{name}}(\text{Sort}_{\text{TID}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \text{TID}])))))$$

An alternative to this plan is not to sort on the original indexed attribute (age in our example), but to introduce a new attribute that holds the rank in the sequence derived from the index scan. This leads to the plan

$$\begin{array}{l}\Pi_{\texttt{name,age}}(\\ \quad \text{Sort}_{\texttt{rank}}(\\ \quad\quad \chi_{e:*(\text{TID}),\texttt{name}:e.\texttt{name}}(\\ \quad\quad\quad \text{Sort}_{\text{TID}}(\\ \quad\quad\quad\quad \chi_{\texttt{rank:counter++}}(\\ \quad\quad\quad\quad\quad \texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \text{TID}])))))\end{array}$$

This alternative might turn out to be more efficient since sorting on an attribute with a dense domain can be implemented efficiently. (We admit that in the above example this is not worth considering.) There is another important application of this technique: XQuery often demands output in document order. If this order is destroyed during processing, it must at the latest be restored when the output it produced [577]. Depending on the implementation (i.e. the representation of document nodes or their identifiers), this might turn out to be a very expensive operation.

The fact that index scans on B-trees return their result ordered on the indexed attributes is also very useful if a merge-join on the same attributes (or a prefix thereof, see Chapter 24 for further details) occurs. An example follows later on.

Some *predicates* are not index SARGable, but can still be evaluated with the index as in the following query

select name
from Emp
where age $\geq$ 25 **and** age $\leq$ 35 **and** age $\neq$ 30

The predicate age $\neq$ 30 is an example of a *residual predicate*. We can once more extend the index scan and compile the query into

$$\Pi_{\texttt{name}}(\chi_{t:x.\text{TID},e:*t,\texttt{name:e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \texttt{age} \neq 30]))$$

Some index scan implementations allow exclusive bounds for start and stop conditions. With them, the query

select name
from Emp
where age $>$ 25 **and** age $<$ 35

can be evaluated using

$$\Pi_{\texttt{name}}(\chi_{t:x.\text{TID},e:*t,\texttt{name:e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 < \texttt{age}; \texttt{age} < 35]))$$

If this is not the case, two residual predicates must be used as in

$$\Pi_{\texttt{name}}(\chi_{t:x.\texttt{TID},e:*t,\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{age}}[x; 25 \leq \texttt{age}; \texttt{age} \leq 35; \texttt{age} \neq 25, \texttt{age} \neq 35]))$$

Especially for predicates on strings, this might be expensive.

Start and stop conditions are optional. To evaluate

select name
from Emp
where age $\geq$ 60

we use $\texttt{age} \geq 60$ as the start condition to find the leaf page containing the key 60. From there on, we scan all the leaf pages "to the right".

If we have no start condition, as in

select name
from Emp
where age $\leq$ 20

we descend the B-tree to the "leftmost" page, i.e. the page containing the smallest key value, and then proceed scanning leaf pages until we encounter the key 20.

Having neither a start nor stop condition is also quite useful. The query

select count(*)
from Emp

can be evaluated by counting the entries in the leaf pages of a B-tree. Since a B-tree typically occupies far fewer pages than the original relation, we have a viable alternative to a relation scan. The same applies to the aggregate functions `sum` and `avg`. The other aggregate functions `min` and `max` can be evaluated much more efficiently by descending to the leftmost or rightmost leaf page of a B-tree. This can be used to answer queries like

select min/max(salary)
from Emp

much more efficiently than by a relation scan. Consider the query

select name
from Emp
where salary = (**select** max(salary)
from Emp)

It can be evaluated by first computing the maximum salary and then retrieving the employees earning this salary. This requires two descendants into the B-tree, while obviously one is sufficient. Depending on the implementation of the index (scan), we might be able to perform this optimization.

Further, the result of an index scan, whether it uses start and/or stop conditions or not, is always sorted on the key. This property can be useful for queries with no predicates. If we have neither a start nor a stop condition, the resulting scan is called *full index scan*. As an example consider the query

select salary
from Emp
order by salary

which is perfectly answered by the following full index scan:

$$\texttt{Emp}_{\texttt{salary}}$$

So far, we have only seen indexes on numerical attributes.

select name, salary
from Emp
where name $\geq$ 'Maaa'

gives rise to a start condition $'\texttt{Maaa}' \leq \texttt{name}$. From the query

select name, salary
from Emp
where name **like** 'M%'

we can deduce the start condition $'\texttt{M}' \leq \texttt{name}$.

To express all the different alternatives of index usage, we need a powerful (and runtime system dependent) index scan expression. Let us first summarize what we can specify for an index scan:

1. the name of the variable for index entries (or pointers to them),
2. the start condition,
3. the stop condition,
4. a residual predicate, and
5. a projection list.

A projection list has entries of the form $a : x.b$ for attribute names a and b and x being the name of the variable for the index entry. $a : x.a$ is also allowed and often abbreviated as a. We also often summarize start and stop conditions into a single expression like in $25 \leq \texttt{age} \leq 35$.

For a full index specification, we list all items in the subscript of the index name separated by a semicolon. Still, we need some extensions to express the queries with aggregation. Let a and b be attribute names, then we allow entries of the form $b : \texttt{aggr}(a)$ in the projection list and start/stop conditions of the form $\texttt{min}/\texttt{max}(a)$. The latter tells us to minimize/maximize the value of the indexed attribute a. Only a complete enumeration gives us the full details. On the other hand, extracting start and stop conditions and residual predicates from a given boolean expression is rather simple. Hence, we often summarize these three under a single predicate. This is especially useful when talking about index scans in general. If we have a full index scan, we leave out the predicate. We use a star '*' as an abbreviated projection list that projects all attributes of the index. (So far, these are the key attribute and the TID.) If the projection list is empty, we assume that only the variable/attribute holding the pointer to the index entry is projected.

Using this notation, we can express some plan fragments. These fragments are complete plans for the above queries, except that the final projection is not present. As an example, consider the following fragment:

$$\chi_{e:*\text{TID},\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{salary}}[x;\text{TID},\texttt{salary}])$$

All the plan fragments seen so far are examples of access paths. An *access path* is a plan fragment with building blocks concerning a single database item. Hence, every building block is an access path. The above plans touch two database items: a relation and an index on some attribute of that relation. If we say that an index concerns the relation it indexes, such a fragment is an access path. For relational systems, the most general case of an access path uses several indexes to retrieve the tuples of a single relation. We will see examples of these more complex access paths in the following section. An access to the original relation is not always necessary. A query that can be answered solely by accessing indexes is called an *index only query*.

A query with `in` like

select name
from Emp
where age in {28, 29, 31, 32}

can be evaluated by taking the minimum and the maximum found in the left-hand side of `in` as the start and stop conditions. We further need to construct a residual predicate. The residual predicate can be represented either as $\texttt{age} = 28 \vee \texttt{age} = 29 \vee \texttt{age} = 31 \vee \texttt{age} = 32$ or as $\texttt{age} \neq 30$.

An alternative is to use a d-join. Consider the example query

select name
from Emp
where salary in {1111, 11111, 111111}

Here, the numbers are far apart and separate index accesses might make sense. Therefore, let us create a temporary relation `Sal` equal to $\{[s : 1111], [s : 11111], [s : 111111]\}$. When using it, the access path becomes

$$\texttt{Sal}[S] < \chi_{e:*\text{TID},\texttt{name}:\texttt{e.name}}(\texttt{Emp}_{\texttt{salary}}[x;\texttt{salary} = S.s;\text{TID}]) >$$

Some B-tree implementations allow efficient searches for multiple ranges and implement *gap skipping* [34, 35, 159, 313, 314, 465, 529]. *Gap skipping*, sometimes also called *zig-zag skipping*, continues the search for keys in a new key range from the latest position visited. The implementation details vary but the main idea of it is that after one range has been completely scanned, the current (leaf) page is checked for its highest key. If it is not smaller than the lower bound of the next range, the search continues in the current page. If it is smaller than the lower bound of the next range, alternative implementations are described in the literature. The simplest is to start a new search from the root for the lower bound. Another alternative uses parent pointers to go up a page as long as the highest key of the current page is smaller than the lower bound of the next range. If this is no longer the case, the search continues downwards again.

Gap skipping gives even more opportunities for index scans and allows efficient implementations of various index nested loop join strategies.

4.12.2 Complex Keys and Data Attributes

In general, an index can have a complex key comprised of the key attributes $k_1, \ldots, k_n$ and the data attributes $d_1, \ldots, d_m$. One possibility is to use a full index scan on such an index. Having more attributes in the index makes it more probable that queries are index-only.

Besides a full index scan, the index can be descended to directly search for the desired tuple(s). Let us take a closer look at this possibility.

If the search predicate is of the form

$$k_1 = c_1 \wedge k_2 = c_2 \wedge \ldots \wedge k_j = c_j$$

for some constants c_i and some $j <= n$, we can generate the start and stop condition

$$k_1 = c_1 \wedge \ldots \wedge k_j = c_j.$$

This simple approach is only possible if the search predicates define values for all search key attributes, starting from the first search key and then for all keys up to the j-th search key with no key attribute unspecified in between. Predicates concerning the other key attributes after the first non-specified key attribute and the additional data attributes only allow for residual predicates. This condition is often not necessary for multi-dimensional index structures, whose discussion is beyond the book.

With ranges things become more complex and highly dependent on the implementation of the facilities of the B-tree. Consider a query predicate restricting key values as follows

$$k_1 = c_1 \wedge k_2 \geq c_2 \wedge k_3 = c_3$$

Obviously, we can generate the start condition $k_1 = c_1 \wedge k_2 \geq c_2$ and the stop condition $k_1 = c_1$. Here, we neglected the condition on k_3 which becomes a residual predicate. However, with some care we can extend the start condition to $k_1 = c_1 \wedge k_2 \geq c_2 \wedge k_3 = c_3$: we only have to keep $k_3 = c_3$ as a residual predicate, since for k_2 values larger than c_2, values different from c_3 can occur for k_3.

If closed ranges are specified for a prefix of the key attributes as in

$$a_1 \leq k_1 \leq b_1 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

we can generate the start key $k_1 = a_1 \wedge \ldots \wedge k_j = a_j$, the stop key $k_1 = b_1 \wedge \ldots \wedge k_j = b_j$, and

$$a_2 \leq k_2 \leq b_2 \wedge \ldots \wedge a_j \leq k_j \leq b_j$$

as the residual predicate. If for some search key attribute k_j the lower bound a_j is not specified, the start condition cannot contain k_j and any k_{j+i}. If for some search key attribute k_j the upper bound b_j is not specified, the stop condition cannot contain k_j and any k_{j+i}.

Two further enhancements of the B-tree functionality possibly allow for alternative start/stop conditions:

- The B-tree implementation allows to specify the order (ascending or descending) for each key attribute individually.

- The B-tree implementation implements forward and backward scans (e.g. implemented in Rdb [34]).

So far, we are only able to exploit query predicates which specify value ranges for a prefix of all key attributes. Consider querying a person on his/her height and his/her hair color: `haircolor = 'blond' and height between 180 and 190`. If we have an index on `sex`, `haircolor`, `height`, this index cannot be used by means of the techniques described so far. However, since there are only the two values `male` and `female` available for `sex`, we can rewrite the query predicate to (`sex = 'm' and haircolor = 'blond' and height between 180 and 190`) `or` (`sex = 'f' and haircolor = 'blond' and height between 180 and 190`) and use two accesses to the index. This approach works fine for attributes with a small domain and is described by Antoshenkov [35]. (See also the previous section for gap skipping.) Since the possible values for key attributes may not be known to the query optimizer, Antoshenkov goes one step further and shifts the construction of search ranges to index scan time. Therefore, the index can be provided with a complex boolean expression which is then refined (rewritten) as soon as search key values become known. Search ranges are then generated dynamically, and gap skipping is applied to skip the intervals between the qualifying ranges during the index scan.

4.13 Multi Index Access Path

We wish to buy a used digital camera and state the following query:

select *
from Camera
where megapixel $>$ 5 **and** distortion $<$ 0.05
and noise $<$ 0.01
and zoomMin $<$ 35 **and** zoomMax $>$ 105

We assume that on every attribute used in the `where` clause there exists an index. Since the predicates are conjunctively connected, we can use a technique called *index and-ing*. Every index scan returns a set (list) of tuple identifiers. These sets/lists are then intersected. This operation is also called *And merge* [550]. Using index and-ing, a possible plan is

$$
\begin{array}{l}
((((\\
\quad \texttt{Camera}_{\texttt{megapixel}}[c; \texttt{megapixel} > 5; \text{TID}]\\
\quad \cap\\
\quad\quad \texttt{Camera}_{\texttt{distortion}}[c; \texttt{distortion} < 0.05; \text{TID}])\\
\quad\quad \cap\\
\quad\quad\quad \texttt{Camera}_{\texttt{noise}}[c; \texttt{noise} < 0.01; \text{TID}])\\
\quad\quad\quad \cap\\
\quad\quad\quad\quad \texttt{Camera}_{\texttt{zoomMin}}[c; \texttt{zoomMin} < 35; \text{TID}])\\
\quad\quad\quad\quad \cap\\
\quad\quad\quad\quad\quad \texttt{Camera}_{\texttt{zoomMax}}[c; \texttt{zoomMax} > 105; \text{TID}])
\end{array}
$$

This results in a set of tuple identifiers that only needs to be dereferenced to access the according `Camera` tuples and produce the final result.

Since the costs of the expression clearly depend on the costs of the index scans and the size of the intermediate TID sets, two questions arise:

- In which order do we intersect the TID sets resulting from the index scans?
- Do we really apply all indexes before dereferencing the tuple identifiers?

The answer to the latter question is clearly *"no"*, if the next index scan is more expensive than accessing the records in the current TID list. It can be shown that the indexes in the cascade of intersections are ordered on increasing $(f_i - 1)/c_i$ terms, where f_i is the selectivity of the index and c_i its access cost. Further, we can stop as soon as accessing the original tuples in the base relation becomes cheaper than intersecting with another index and subsequently accessing the base relation. EX

Index or-ing is used to process disjunctive predicates. Here, we take the *union* of the TID sets to produce a set of TIDs containing references to all qualifying tuples. Note that duplicates must be eliminated during the processing of the union. This operation is also called *Or merge* [550]. Consider the query

select *
from Emp
where yearsOfEmployment ≥ 30
or age ≥ 65

This query can be answered by constructing a TID set using the expression

$$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \geq 30; \text{TID}] \cup \texttt{Emp}_{\texttt{age}}[c; \texttt{age} \geq 65; \text{TID}]$$

and then dereferencing the list of tuple identifiers. Again, the index accessing can be ordered for better performance. Given a general boolean expression in `and` and `or`, constructing the optimal access path using index and-ing and or-ing is a challenging task that will be discussed in Chapter **??**. This task is even more challenging, if some simple predicates occur more than once in the complex boolean expression and factorization has to be taken into account. This issue was first discussed by Chaudhuri, Ganesan and Saragawi [134]. We will come back to this in Chapter **??**.

The names *index and-ing* and *or-ing* become clear if bitmap indexes are considered. Then the bitwise `and` and `or` operations can be used to efficiently compute the intersection and union. ToDo

Excursion on bitmap indexes. □

There are even more possibilities to work with TID sets. Consider the query

select *
from Emp
where yearsOfEmployment $\neq 10$
and age ≥ 65

This query can be evaluated by scanning the index on `age` and then eliminating all employees with `yearsOfEmployment = 10`:

$$\texttt{Emp}_{\texttt{age}}[c; \texttt{age} \geq 65; \text{TID}] \setminus \texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \neq 10; \text{TID}]$$

Let us call the application of set difference on index scan results *index differencing*.

Some predicates might not be very restrictive in the sense that more than half the index has to be scanned. By negating these predicates and using index differencing, we can make sure that at most half of the index needs to be scanned. As an example consider the query

select *
from Emp
where yearsOfEmployment ≤ 5
and age ≤ 65

Assume that most of our employees' age is below 65. Then

$$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \leq 5; \text{TID}] \setminus \texttt{Emp}_{\texttt{age}}[c; \texttt{age} > 65; \text{TID}]$$

could be more efficient than

$$\texttt{Emp}_{\texttt{yearsOfEmployment}}[c; \texttt{yearsOfEmployment} \leq 5; \text{TID}] \cap \texttt{Emp}_{\texttt{age}}[c; \texttt{age} \leq 65; \text{TID}]$$

4.14 Indexes and Joins

There are two issues when discussing indexes and joins. The first is that indexes can be used to speed up join processing. The second is that index accesses can be expressed as joins. We discuss both of these issues, starting with the latter.

In our examples, we used the map operation to (implicitly) access the relation by dereferencing the tuple identifiers. We can make the implicit access explicit by exchanging the map operator by a d-join or even a join. Then, for example,

$$\chi_{e:*\text{TID},\texttt{name:e.name}}(\texttt{Emp}_{\texttt{salary}}[x; 25 \leq \texttt{age} \leq 35; \text{TID}])$$

becomes

$$\texttt{Emp}_{\texttt{salary}}[x; 25 \leq \texttt{age} \leq 35; \text{TID}] < \chi_{e:*\text{TID},\texttt{name:e.name}}(\Box) >$$

where $\Box$ returns a single empty tuple. Assume that every tuple contains an attribute TID containing its TID. This attribute does not have to be stored explicitly but can be derived. Then, we have the following alternative access path for the join (ignoring projections):

$$\texttt{Emp}_{\texttt{salary}}[x; 25 \leq \texttt{age} \leq 35] \bowtie_{x.\text{TID}=e.\text{TID}} \texttt{Emp}[e]$$

For the join operator, the *pointer-based join* implementation developed in the context of object-oriented databases may be the most efficient way to evaluate the access path [774]. Obviously, sorting the result of the index scan on the tuple identifiers can speed up processing since it turns random into sequential I/O. However, this destroys the

ToDo

order on the key which might itself be useful later on during query processing or required by the query[7]. Sorting the tuple identifiers was proposed by, e.g., Yao [923], Makinouchi, Tezuka, Kitakami, and Adachi in the context of RDB/V1 [564]. The different variants (whether or not and where to sort, join order) can now be transparently determined by the plan generator: no special treatment is necessary. Further, the join predicates can not only be on the tuple identifiers but also on key attributes. This often allows to join with other than the indexed relations (or their indexes) before accessing the relation.

Rosenthal and Reiner proposed to use joins to represent access paths with indexes [707]. This approach is very elegant since no special treatment for index processing is required. However, if there are many relations and indexes, the search space might become very large, as every index increases the number of joins to be performed. This is why Mohan, Haderle, Wang, and Cheng abondoned this approach and sketched a heuristics which determines an access path in case multiple indexes on a single table exist [603].

The query

select name,age
from Person
where name like 'R%' and age between 40 and 50

is an index only query (assuming indexes on `name` and `age`) and can be translated to

$$\Pi_{\texttt{name,age}}(\texttt{Emp}_{\texttt{age}}[a; 40 \leq \texttt{age} \leq 50; \texttt{TIDa}, \texttt{age}] \bowtie_{\texttt{TIDa=TIDn}} \texttt{Emp}_{\texttt{name}}[n; \texttt{name} \geq' R'; \texttt{name} \leq' R'; \texttt{TIDn}, \texttt{name}])$$

Let us now discuss the former of the two issues mentioned in the section's introduction. The query

select *
from Emp e, Dept d
where e.name = 'Maier' and e.dno = d.dno

can be directly translated to

$$\sigma_{e.\texttt{name}=\texttt{'Maier'}}(\texttt{Emp}[e]) \bowtie_{e.\texttt{dno}=d.\texttt{dno}} \texttt{Dept}[d]$$

If there are indexes on `Emp.name` and `Dept.dno`, we can replace $\sigma_{e.\texttt{name}=\texttt{'Maier'}}(\texttt{Emp}[e])$ by an index scan as we have seen previously:

$$\chi_{e:*(x.TID),\mathcal{A}(\texttt{Emp}):e.*}(\texttt{Emp}_{\texttt{name}}[x; \texttt{name} = \texttt{'Maier'}])$$

Here, $\mathcal{A}(\texttt{Emp}) : t.*$ abbreviates access to all `Emp` attributes. This especially includes `dno:t.dno`. (Strictly speaking, we do not have to access the `name` attribute, since its value is already known.)

[7] Restoring the order may be cheaper than typical sorting since tuples can be numbered before the first sort on tuple identifiers, and this dense numbering leads to efficient sort algorithms.

As we have also seen, an alternative is to use a d-join instead:

$$\texttt{Emp}_{\texttt{name}}[x; \texttt{name} = \text{`}\texttt{Maier}'] < \chi_{t:*(x.TID),\mathcal{A}(e)t.*}(\Box) >$$

Let us abbreviate $\texttt{Emp}_{\texttt{name}}[x; \texttt{name} = \text{`}\texttt{Maier}']$ by E_i and $\chi_{t:*(x.TID),\mathcal{A}(e)t.*}(\Box)$ by E_a.

Now, for any `e.dno`, we can use the index on `Dept.dno` to access the according department tuple:

$$E_i < E_a >< \texttt{Dept}_{\texttt{dno}}[y; y.dno = dno] >$$

Note that the inner expression $\texttt{Dept}_{\texttt{dno}}[y; y.dno = dno]$ contains the free variable `dno`, which is bound by E_a. Dereferencing the TID of the department results in the following algebraic modelling which models a complete *index nested loop join*:

$$E_i < E_a >< \texttt{Dept}_{\texttt{dno}}[y; y.\texttt{dno} = \texttt{dno}; \texttt{dTID} : y.\text{TID}] >< \chi_{u:*\texttt{dTID},\mathcal{A}(\texttt{Dept})u.*}(\Box) >$$

Let us abbreviate $\texttt{Dept}_{\texttt{dno}}[y; y.\texttt{dno} = \texttt{dno}; \texttt{dTID} : y.\text{TID}]$ by D_i and $\chi_{u:*\texttt{dTID},\mathcal{A}(\texttt{Dept})u.*}(\Box)$ by D_a. Fully abbreviated, the expression then becomes

$$E_i < E_a >< D_i >< D_a >$$

Several optimizations can possibly be applied to this expression. Sorting the *outer* of a d-join is useful under several circumstances since it may

- turn random I/O into sequential I/O and/or
- avoid reading the same page twice.

In our example expression,

- we can sort the result of expression E_i on `TID` in order to turn random I/O into sequential I/O, if there are many employees named "Maier".
- we can sort the result of the expression $E_i < E_a >$ on `dno` for two reasons:
  - If there are duplicates for `dno`, i.e. there are many employees named "Maier" in each department, then this guarantees that no index page (of the index `Dept.dno`) has to be read more than once.
  - If additionally `Dept.dno` is a clustered index or `Dept` is an index-only table contained in `Dept.dno`, then large parts of the random I/O can be turned into sequential I/O.
  - If the result of the inner is materialized (see below), then only one result needs to be stored. Note that sorting is not necessary, but grouping would suffice to avoid duplicate work.
- We can sort the result of the expression $E_i < E_a >< D_i >$ on `dTID` for the same reasons as mentioned above for sorting the result of E_i on `TID`.

EX The reader is advised to explicitly write down the alternatives. Another exercise is to give plan alternatives for the different cases of DB2's Hybrid Join [313] which can now be decomposed into primitives like relation scan, index scan, d-join, sorting, TID dereferencing, and access to a unique index (see below).

Let us take a closer look at materializating the result of the inner of the d-join. IBM's DB2 for MVS considers temping (i.e. creating a temporary relation) the inner if it is an index access [313]. Graefe provides a general discussion on the subject [342]. Let us start with the above example. Typically, many employees will work in a single department and possibly several of them are called "Maier". For everyone of them, we can be sure that there exists at most one department. Let us assume that referential integrity has been specified. Then, there exists exactly one department for every employee. We have to find a way to rewrite the expression

$$E_i < E_a >< \texttt{Dept}_{\texttt{dno}}[y; y.\texttt{dno} = \texttt{dno}; \texttt{dTID} : y.\text{TID}] >$$

such that the mapping $\texttt{dno} \longrightarrow \texttt{dTID}$ is explicitly materialized (or, as one could also say, *cached*). For this purpose, Hellerstein and Naughton introduced a modified version of the map operator that materializes its result [405]. Let us denote this operator by $\chi^{\texttt{mat}}$. The advantage of using this operator is that it is quite general and can be used for different purposes (see e.g. [94], Chap. **??**, Chap. **??**). Since the map operator extends a given input tuple by some attribute values, which must be computed by an expression, we need one to express the access to a unique index. For our example, we write

$$\texttt{IdxAcc}^{Dept}_{dno}[y; y.\texttt{dno} = \texttt{dno}]$$

to express the lookup of a single (unique) entry in the index on `Dept.dno`. We assume that the result is a (pointer to the) tuple containing the key attributes and all data attributes including the TID of some tuple. Then, we have to perform a further attribute access (dereferenciation) if we are interested in only one of the attributes.

Now, we can rewrite the above expression to

$$E_i < E_a >< \chi^{\texttt{mat}}_{dTID:(\texttt{IdxAcc}^{Dept}_{dno}[y;y.\texttt{dno}=\texttt{dno}]).\text{TID}}(\Box) >$$

If we further assume that the outer ($E_i < E_a >$) is sorted on `dno`, then it suffices to remember only the TID for the latest `dno`. We define the map operator $\chi^{\texttt{mat},1}$ to do exactly this. A more efficient plan could thus be

$$\text{Sort}_{\texttt{dno}}(E_i < E_a >) < \chi^{\texttt{mat},1}_{dTID:(\texttt{IdxAcc}^{Dept}_{dno}[y;y.\texttt{dno}=\texttt{dno}]).\text{TID}}(\Box) >$$

where, strictly speaking, sorting is not necessary: grouping would suffice.

Consider a general expression of the form $e_1 < e_2 >$. The free variables used in e_2 must be a subset of the variables (attributes) produced by e_1, i.e. $\mathcal{F}(e_2) \subseteq \mathcal{A}(e_1)$. Even if e_1 does not contain duplicates, the projection of e_1 on $\mathcal{F}(e_2)$ may contain duplicates. If so, materialization could pay off. However, in general, for every binding of the variables $\mathcal{F}(e_2)$, the expression e_2 may produce several tuples. This means that using $\chi^{\texttt{mat}}$ is not sufficient. Consider the query

select *
from Emp e, Wine w
where e.yearOfBirth = w.year

If we have no indexes, we can answer this query by a simple join where we only have to decide the join method and which of the relations becomes the outer and which the inner. Assume we have only wines from a few years. (Alternatively, some selection could have been applied.) Then it might make sense to consider the following alternative:

$$\texttt{Wine}[w] < \sigma_{e.\texttt{yearOfBirth}=w.\texttt{year}}(\texttt{Emp}[e]) >$$

However, the relation `Emp` is scanned once for each `Wine` tuple. Hence, it might make sense to materialize the result of the inner for every `year` value of `Wine` if we have only a few `year` values. In other words, if we have many duplicates for the `year` attribute of `Wine`, materialization may pay off since then we have to scan `Emp` only once for each `year` value of `Wine`. To achieve caching of the inner, in case every binding of its free variables possibly results in many tuples, requires a new operator. Let us call this operator *memox* and denote it by $\mathfrak{M}$ [342, 94]. For the free variables of its only argument, it remembers the set of result tuples produced by its argument expression and does not evaluate it again if it is already cached. Using memox, the above plan becomes

$$\texttt{Wine}[w] < \mathfrak{M}(\sigma_{\texttt{e.yearOfBirth}=\texttt{w.year}}(\texttt{Emp}[e])) >$$

It should be clear that for more complex inners, the memox operator can be applied at all branches, giving rise to numerous caching strategies. Analogously to the materializing map operator, we are able to restrict the materialization to the results for a single binding for the free variables if the outer is sorted (or grouped) on the free variables:

$$\text{Sort}_{w.\texttt{yearOfBirth}}(\texttt{Wine}[w]) < \mathfrak{M}^1(\sigma_{\texttt{e.yearOfBirth}=\texttt{w.year}}(\texttt{Emp}[e])) >$$

Things can become even more efficient if there is an index on `Emp.yearOfBirth`:

$$\begin{array}{l}\text{Sort}_{w.\texttt{yearOfBirth}}(\texttt{Wine}[w])\\ < \mathfrak{M}^1(\texttt{Emp}_{\texttt{yearOfBirth}}[x; \texttt{x.yearOfBirth} = \texttt{w.year}] < \chi_{e:*(x.\text{TID}),\mathcal{A}(\texttt{Emp}):*e}(\Box) >) >\end{array}$$

So far we have seen different operators which materialize values: Tmp, $\mathfrak{M}$, and $\chi_{\texttt{mat}}$. The latter in two variants. As an exercise, the reader is advised to discuss the differences between them. EX

Assume, we have indexes on both `Emp.yearOfBirth` and `Wine.year`. Besides the possibilities to use either `Emp` or `Wine` as the outer, we now also have the possibility to perform a join on the indexes before accessing the actual `Emp` and `Wine` tuples. Since the index scan produces its output ordered on the key attributes, a simple merge join suffices (and we are back at the latter):

$$\texttt{Emp}_{\texttt{yearOfBirth}}[x] \bowtie^{\texttt{merge}}_{x.\texttt{yearOfBirth}=y.\texttt{year}} \texttt{Wine}_{\texttt{year}}[y]$$

This example makes clear that the order provided by an index scan can be used to speed up join processing. After evaluating this plan fragment, we have to access the actual `Emp` and `Wine` tuples. We can consider zero, one, or two sorts on their respective tuple identifiers. If the join is sufficiently selective, one of these alternatives may prove more sufficient than the ones we have considered so far. EX

4.15 Remarks on Access Path Generation

A last kind of optimization we briefly want to mention is *sideways information passing*. Consider a simple join between two relations: $R \bowtie_{R.a=S.b} S$. If we decide to perform a sort merge join or a hash join, we can implement it by first sorting/partitioning R before looking at S. While doing so, we can remember the minimum and maximum value of `R.a` and use these as a restriction on S such that fewer tuples of S have to be sorted/partitioned. In case we perform a blockwise nested loop join, after the first scan of S we know the minimum and maximum value of `S.b` and can use these to restrict R.

If the number of distinct values of `R.a` is small, we could also decide to remember all these values and evaluate perform a semi-join before the actual join. Algebraically, this could be expressed as

$$R \bowtie_{R.a=S.b} (S \succ\!\!\lhd_{S.b=R.a} \Pi_{R.a}(R))$$

An alternative is to use a bitmap to represent the projection of R on a.

The semi-join technique should be well-known from distributed database systems. In deductive database systems, this kind of optimization often carries the attribute *magic*. We will more deeply discuss this issue in Chapter **??**.

The following problem is not discussed in the book. Assume that we have fully partitioned a relation vertically into a set of files which are chronologically ordered. Then, the attribute a_i of the j-th tuple can be found at the j-th position of the i-th file. This organizion is called *partitioned transposed file* [52]. (Compare this with variant (projection) indexes [625] and small materialized aggregates [598].) The problem is to find an access strategy to all the attribute required by the query given a collection of restriction on some of the relation's attributes. This problem has been discussed in depth by Batory [52]. Full vertical partitioning is also used as the organizing principle of Monet []. Lately, it also gained some interest in the US [].

4.16 Counting the Number of Accesses

4.16.1 Counting the Number of Direct Accesses

After the index scan, we have a set of (distinct) tuple identifiers for which we have to access the original tuples. The question we would like to answer is:

> How many pages do we have to read?

Let R be the relation for which we have to retrieve the tuples. Then we use the following abbreviations

N	$\|R\|$	number of tuples in the relation R
m	$\|\|R\|\|$	number of pages on which tuples of R are stored
B	N/m	number of tuples per page (*blocking factor*)
k		number of (distinct) TIDs for which tuples have to be retrieved

We assume that the tuples are uniformly distributed among the m pages. Then, each page stores $B = N/m$ tuples. B is called *blocking factor*.

Let us consider some borderline cases. If $k > N - N/m$ or $m = 1$, then all pages are accessed. If $k = 1$ then exactly one page is accessed. The answer to the general question will be expressed in terms of *buckets* (pages in the above case) and *items* contained therein (tuples in the above case). Later on, we will also use extents, cylinders, or tracks as buckets and tracks or sectors/blocks as items.

We assume that a bucket contains items. The total number of items will be N and the number of requested items will be k. The above question can then be reformulated to how many buckets contain at least one of the k requested items, i.e. how many qualifying buckets exist. We start out by investigating the case where the items are uniformly distributed among the buckets. Two subcases will be distinguished:

1. k distinct items are requested

2. k non-distinct items are requested.

We then discuss the case where the items are non-uniformly distributed.

In any case, the underlying access model is random access. For example, given a tuple identifier, we can directly access the page storing the tuple. Other access models are possible. The one we will subsequently investigate is sequential access where the buckets have to be scanned sequentially in order to find the requested items. After that, we are prepared to develop a model for disk access costs.

Throughout this section, we will further assume that the probability that we request a set with k items is $\frac{1}{\binom{N}{k}}$ for all of the $\binom{N}{k}$ possibilities to select a k-set.[8] We often make use of established equalities for binomial coefficients. For convenience, the most frequently used equalities are listed in Appendix E.

Selecting k distinct items

Our first theorem was discovered independently by Waters [878] and Yao [920]. We formulate it in terms of buckets containing items. We say a bucket *qualifies* if it contains at least one of the k items we are looking for.

Theorem 4.16.1 (Waters/Yao) *Consider m buckets with n items each. Then there is a total of $N = nm$ items. If we randomly select k distinct items from all items, then the number of qualifying buckets is*

$$\overline{\mathcal{Y}}_n^{N,m}(k) = m * \mathcal{Y}_n^N(k) \tag{4.2}$$

where $\mathcal{Y}_n^N(k)$ is the probability that a bucket contains at least one item. This probability is equal to

$$\mathcal{Y}_n^N(k) = \begin{cases} [1-p] & k \leq N-n \\ 1 & k > N-n \end{cases}$$

where p is the probability that a bucket contains none of the k items. The following

[8] A k-set is a set with cardinality k.

alternative expressions can be used to calculate p:

$$p = \frac{\binom{N-n}{k}}{\binom{N}{k}} \tag{4.3}$$

$$= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i} \tag{4.4}$$

$$= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i} \tag{4.5}$$

The second expression (4.4) is due to Yao, the third (4.5) is due to Waters. Palvia and March proved both formulas to be equal [642] (see also [38]). The fraction $m = N/n$ may not be an integer. For these cases, it is advisable to have a Gamma-function based implementation of binomial coeffcients at hand (see [673] for details).

Depending on k and n, either the expression of Yao or the one of Waters is faster to compute. After the proof of the above formulas and the discussion of some special cases, we will give several approximations for p.

Proof The total number of possibilities to pick the k items from all N items is $\binom{N}{k}$. The number of possibilities to pick k items from all items not contained in a fixed single bucket is $\binom{N-n}{k}$. Hence, the probability p that a bucket does not qualify is $p = \binom{N-n}{k}/\binom{N}{k}$. Using this result, we can do the following calculation

$$\begin{aligned} p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \\ &= \frac{(N-n)!\ k!(N-k)!}{k!((N-n)-k)!\ N!} \\ &= \prod_{i=0}^{k-1} \frac{N-n-i}{N-i} \end{aligned}$$

which proves the second expression. The third follows from

$$\begin{aligned} p &= \frac{\binom{N-n}{k}}{\binom{N}{k}} \\ &= \frac{(N-n)!\ k!(N-k)!}{k!((N-n)-k)!\ N!} \\ &= \frac{(N-n)!\ (N-k)!}{N!\ ((N-k)-n)!} \\ &= \prod_{i=0}^{n-1} \frac{N-k-i}{N-i} \end{aligned}$$

□

Let us list some special cases:

If	then $\mathcal{Y}_m^N(k) =$
$n = 1$	k/N
$n = N$	1
$k = 0$	0
$k = 1$	$B/N = (N/m)N = 1/m$
$k = N$	1

We examine a slight generalization of the first case in more detail. Let N items be distributed over N buckets such that every bucket contains exactly one item. Further let us be interested in a subset of m buckets ($1 \leq m \leq N$). If we pick k items, then the number of buckets within the subset of size m that qualify is

$$m\mathcal{Y}_1^N(k) = m\frac{k}{N} \tag{4.6}$$

In order to see that the two sides are equal, we perform the following calculation:

$$\begin{aligned}
\mathcal{Y}_1^N(k) &= (1 - \frac{\binom{N-1}{k}}{\binom{N}{k}}) \\
&= (1 - \frac{\frac{(N-1)!}{k!((N-1)-k)!}}{\frac{N!}{k!(N-k)!}}) \\
&= (1 - \frac{(N-1)!k!(N-k)!}{N!k!((N-1)-k)!}) \\
&= (1 - \frac{N-k}{N}) \\
&= (\frac{N}{N} - \frac{N-k}{N}) \\
&= \frac{N-N+k}{N} \\
&= \frac{k}{N}
\end{aligned}$$

Since the computation of $\mathcal{Y}_n^N(k)$ can be quite expensive, several approximations have been developed. The first one was given by Waters [877, 878]:

$$p \approx (1 - k/N)^n$$

This approximation (also described elsewhere [308, 642]) turns out to be pretty good. However, below we will see even better approximations.

For $\overline{\mathcal{Y}}_n^{N,m}(k)$ Whang, Wiederhold, and Sagalowicz gave the following approximation for faster calculation [889]:

$$\begin{aligned}
m * [\quad & (1 - (1 - 1/m)^k) + \\
& (1/(m^2 b) * k(k-1)/2 * (1 - 1/m)^{k-1}) + \\
& (1.5/(m^3 p^4) * k(k-1)(2k-1)/6 * (1 - 1/m)^{k-1}) \ ]
\end{aligned}$$

A rough estimate is presented by Bernstein, Goodman, Wong, Reeve, and Rothnie [73]:

$$\overline{\mathcal{Y}}_n^{N,m}(k) \approx \begin{cases} k & \text{if } k < \frac{m}{2} \\ \frac{k+m}{2} & \text{if } \frac{m}{2} \leq k < 2m \\ m & \text{if } 2m \leq k \end{cases}$$

An interesting and useful result was derived by Dihr and Saharia [234]. They give two formulas and show that they are lower and upper bounds to Water and Yao's formula. The upper and lower bounds for p are

$$\begin{aligned} p_{\text{lower}} &= (1 - \frac{k}{N - \frac{n-1}{2}})^n \\ p_{\text{upper}} &= ((1 - \frac{k}{N}) * (1 - \frac{k}{N - n + 1}))^{n/2} \end{aligned}$$

for $n = N/m$. Dihr and Saharia claim that the maximal difference resulting from the use of the lower and the upper bound to compute the number of page accesses is 0.224—far less than a single page access.

Selecting k non-distinct items

So far, we assumed that we retrieve k *distinct* items. We could ask the same question for k *non-distinct* items. This question demands a different urn model. In urn model terminology, the former case is an urn model with a *non-replacement* assumption, while the latter case is one with a *replacement* assumption. (Deeper insight into urn models is given by Drmota, Gardy, and Gittenberger [240].)

Before presenting a theorem discovered by Cheung [164], we repeat a theorem from basic combinatorics. We know that the number of subsets of size k of a set with N elements is $\binom{N}{k}$. The following lemma gives us the number of k-multisets[9].

Lemma 4.16.2 *Let S be a set with $|S| = N$ elements. Then, the number of multisets with cardinality k containing only elements from S is*

$$\binom{N + k - 1}{k}$$

For a proof we just note that there is a bijection between the k-multisets and the k-subsets of a $N + k - 1$-set. We can go from a multiset to a set by f with $f(\{x_1 \leq \ldots \leq x_k\}) = \{x_1 + 0 < x_2 + 1 < \ldots < x_k + (k-1)\}$ and from a set to a multiset via g with $g(\{x_1 < \ldots < x_k\}) = \{x_1 - 0 < x_2 - 1 < \ldots < x_k - (k-1)\}$.

Theorem 4.16.3 (Cheung) *Consider m buckets with n items each. Then there is a total of $N = nm$ items. If we randomly select k not necessarily distinct items from all items, then the number of qualifying buckets is*

$$\overline{\mathit{Cheung}}_n^{N,m}(k) = m * \mathit{Cheung}_n^N(k) \tag{4.7}$$

where

$$\mathit{Cheung}_n^N(k) = [1 - \tilde{p}] \tag{4.8}$$

[9] A k-multiset is a multiset with k elements.

with the following equivalent expressions for $\tilde{p}$:

$$\tilde{p} = \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \tag{4.9}$$

$$= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i} \tag{4.10}$$

$$= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i} \tag{4.11}$$

Eq. 4.9 follows from the observation that the probability that some bucket does not contain any of the k possibly duplicate items is $\frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}}$. Eq. 4.10 follows from

$$\begin{aligned}
\tilde{p} &= \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \\
&= \frac{(N-n+k-1)!\ k!((N+k-1)-k)!}{k!((N-n+k-1)-k)!\ (N+k-1)!} \\
&= \frac{(N-n-1+k)!\ (N-1)!}{(N-n-1)!\ (N-1+k)!} \\
&= \prod_{i=0}^{k-1} \frac{N-n+i}{N+i}
\end{aligned}$$

Eq. 4.11 follows from

$$\begin{aligned}
\tilde{p} &= \frac{\binom{N-n+k-1}{k}}{\binom{N+k-1}{k}} \\
&= \frac{(N-n+k-1)!\ k!((N+k-1)-k)!}{k!((N-n+k-1)-k)!\ (N+k-1)!} \\
&= \frac{(N+k-1-n)!\ (N-1)!}{(N+k-1)!\ (N-1-n)!} \\
&= \prod_{i=0}^{n-1} \frac{N-n+i}{N+k-n+i} \\
&= \prod_{i=0}^{n-1} \frac{N-1-i}{N-1+k-i}
\end{aligned}$$

□

Cardenas discovered a formula that can be used to approximate $\tilde{p}$ [111]:

$$(1-n/N)^k$$

As Cheung pointed out, we can use the theorem to derive the number of non-distinct accessed items contained in a k-multiset.

Corollary 4.16.4 *Let S be a k-multiset containing elements from an N-set T. Then the number of distinct items contained in S is*

$$\mathcal{D}(N,k) = \frac{Nk}{N+k-1} \tag{4.12}$$

if the elements in T occur with the same probability in S.

We apply the theorem for the special case where every bucket contains exactly one item ($n = 1$). In this case, $\prod_{i=0}^{0} \frac{N-1-i}{N-1+k-i} = \frac{N-1}{N-1+k}$. And the number of qualifying buckets is $N(1 - \frac{N-1}{N-1+k}) = N(\frac{N-1+k-N+1}{N-1+k}) = N\frac{k}{N+k-1}$. □

A useful application of this formula is to calculate the size of a projection [164]. Another use is that calculating the number of distinct values contained in a multiset allows us to shift from the model with replacement to a model without replacement. However, there is a difference between

$$\overline{\mathcal{Y}}_n^{N,m}(Distinct(N,k)) \approx \overline{\text{Cheung}}_n^{N,m}(k)$$

even when computing $\overline{\mathcal{Y}}$ with Eq. 4.5. Nonetheless, for $n \geq 5$, the error is less than two percent. One of the problems when calculating the result of the left-hand side is that the number of distinct items is not necessarily an integer. To solve this problem, we can implement all our formulas using the Gamma-function. But even then a small difference remains.

Non-Uniform Distribution of Items

In the previous sections, we assumed that

1. every page contains the same number of records, and
2. every record is accessed with the same probability.

We now turn to relax the first assumption. Christodoulakis models the distribution by m numbers n_i (for $1 \leq i \leq m$) if there are m buckets. Each n_i equals the number of records in some bucket i [168]. Luk proposes Zipfian record distribution [557]. However, Ijbema and Blanken say that Water and Yao's formula is still better, as Luk's formula results in too low values [430]. They all come up with the same general formula presented below. Vander Zander, Taylor, and Bitton [931] discuss the problem of correlated attributes which results in some clusteredness. Zahorjan, Bell, and Sevcik discuss the problem where every item is assigned its own access probability [930]. That is, they relax the second assumption. We will come back to these issues in Section 27.2.

We still assume that every item is accessed with the same probability. However, we relax the first assumption. The following formula derived by Christodoulakis [168], Luk [557], and Ijbema and Blanken [430] is a simple application of Waters's and Yao's formula to a more general case.

Theorem 4.16.5 (Yao/Waters/Christodoulakis) *Assume a set of m buckets. Each bucket contains $n_j > 0$ items ($1 \leq j \leq m$). The total number of items is $N = \sum_{j=1}^{m} n_j$. If we look up k distinct items, then the probability that bucket j qualifies is*

$$\mathcal{W}_{n_j}^{N}(k,j) = [1 - \frac{\binom{N-n_j}{k}}{\binom{N}{k}}] \quad (= \mathcal{Y}_{n_j}^{N}(k)) \tag{4.13}$$

and the expected number of qualifying buckets is

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) := \sum_{j=1}^{m} \mathcal{W}_{n_j}^{N}(k,j) \tag{4.14}$$

Note that the product formulation in Eq. 4.5 of Theorem 4.16.1 results in a more efficient computation. We make a note of this in the following corollary.

Corollary 4.16.6 *Assume a set of m buckets. Each bucket contains $n_j > 0$ items ($1 \leq j \leq m$). The total number of items is $N = \sum_{j=1}^{m} n_j$. If we look up k distinct items, then the expected number of qualifying buckets is*

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{j=1}^{m}(1-p_j) \tag{4.15}$$

with

$$p_j = \begin{cases} \prod_{i=0}^{n_j-1} \frac{N-k-i}{N-i} & k \leq n_j \\ 0 & N-n_j < k \leq N \end{cases} \tag{4.16}$$

If we compute the p_j after we have sorted the n_j in ascending order, we can use the fact that

$$p_{j+1} = p_j * \prod_{i=n_j}^{n_{j+1}-1} \frac{N-k-i}{N-i}.$$

We can also use the theorem to calculate the number of qualifying buckets in case the distribution is given by a histogram.

Corollary 4.16.7 *For $1 \leq i \leq L$ let there be l_i buckets containing n_i items. Then the total number of buckets is $m = \sum_{i=1}^{L} l_i$, and the total number of items in all buckets is $N = \sum_{i=1}^{L} l_i n_i$. For k randomly selected items, the number of qualifying buckets is*

$$\overline{\mathcal{W}}_{n_j}^{N,m}(k) = \sum_{i=1}^{L} l_i \mathcal{Y}_{n_j}^{N}(k) \tag{4.17}$$

Last in this section, let us calculate the probability distribution for the number of qualifying items within a bucket. The probability that $x \leq n_j$ items in a bucket j qualify can be calculated as follows. The number of possibilities to select x items in bucket n_j is $\binom{n_j}{x}$. The number of possibilites to draw the remaining $k-x$ items from the other buckets is $\binom{N-n_j}{k-x}$. The total number of possibilities to distribute k items over the buckets is $\binom{N}{k}$. This shows the following:

Theorem 4.16.8 *Assume a set of m buckets. Each bucket contains $n_j > 0$ items ($1 \leq j \leq m$). The total number of items is $N = \sum_{j=1}^{m} n_j$. If we look up k distinct items, the probability that x items in bucket j qualify is*

$$\mathcal{X}_{n_j}^{N}(k,x) = \frac{\binom{n_j}{x}\ \binom{N-n_j}{k-x}}{\binom{N}{k}} \tag{4.18}$$

Further, the expected number of qualifying items in bucket j is

$$\overline{\mathcal{X}}^{N,m}_{n_j}(k) = \sum_{x=0}^{\min(k,n_j)} x\mathcal{X}^N_{n_j}(k,x) \tag{4.19}$$

In standard statistics books the probability distribution $\mathcal{X}^N_{n_j}(k,x)$ is called *hypergeometric distribution.*

Let us consider the case where all n_j are equal to n. Then we can calculate the average number of qualifying items in a bucket. With $y := \min(k,n)$ we have

$$\begin{aligned}
\overline{\mathcal{X}}^{N,m}_{n_j}(k) &= \sum_{x=0}^{\min(k,n)} x\mathcal{X}^N_n(k,x) \\
&= \sum_{x=1}^{\min(k,n)} x\mathcal{X}^N_n(k,x) \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} x \binom{n}{x}\binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{x}{1}\binom{n}{x}\binom{N-n}{k-x} \\
&= \frac{1}{\binom{N}{k}} \sum_{x=1}^{y} \binom{n}{1}\binom{n-1}{x-1}\binom{N-n}{k-x} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \sum_{x=0}^{y-1} \binom{n-1}{0+x}\binom{N-n}{(k-1)-x} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{n-1+N-n}{0+k-1} \\
&= \frac{\binom{n}{1}}{\binom{N}{k}} \binom{N-1}{k-1} \\
&= n\frac{k}{N} = \frac{k}{m}
\end{aligned}$$

Let us consider the even more special case where every bucket contains a single item. That is, $N = m$ and $n_i = 1$. The probability that a bucket contains a qualifying item reduces to

$$\begin{aligned}
\mathcal{X}^N_1(k,x) &= \frac{\binom{1}{x}\ \binom{N-1}{k-1}}{\binom{N}{k}} \\
&= \frac{\binom{N-1}{k-1}}{\binom{N}{k}} \\
&= \frac{k}{N}\ (= \frac{k}{m})
\end{aligned}$$

Since x can then only be zero or one, the average number of qualifying items a bucket contains is also $\frac{k}{N}$.

The formulas presented in this section can be used to estimate the number of block/page accesses in case of random direct accesses. As we will see next, other kinds of accesses occur and need different estimates.

4.16.2 Counting the Number of Sequential Accesses

Vector of Bits

When estimating seek costs, we need to calculate the probability distribution for the distance between two subsequent qualifying cylinders. We model the situation as a bitvector of length B with b bits set to 1. Then B corresponds to the number of cylinders and a 1 indicates that a cylinder qualifies.

Theorem 4.16.9 *Assume a bitvector of length B. Within it b ones are uniformly distributed. The remaining $B - b$ bits are zero. Then the probability distribution of the number j of zeros*

1. *between two consecutive ones,*
2. *before the first one, and*
3. *after the last one*

is given by

$$\mathcal{B}_b^B(j) = \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \tag{4.20}$$

A more general theorem (see Theorem 4.16.13) was first presented by Yao [921]. The above formulation is due to Christodoulakis [175].

To see why the formula holds, consider the total number of bitvectors having a one in position i followed by j zeros followed by a one. This number is $\binom{B-j-2}{b-2}$. We can chose $B - j - 1$ positions for i. The total number of bitvectors is $\binom{B}{b}$ and each bitvector has $b-1$ sequences of the form that a one is followed by a sequence of zeros is followed by a one. Hence,

$$\begin{aligned} \mathcal{B}_b^B(j) &= \frac{(B-j-1)\binom{B-j-2}{b-2}}{(b-1)\binom{B}{b}} \\ &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \end{aligned}$$

Part 1. of the theorem follows. To prove part 2., we count the number of bitvectors that start with j zeros before the first one. There are $B - j - 1$ positions left for the remaining $b - 1$ ones. Hence, the number of these bitvectors is $\binom{B-j-1}{b-1}$ and part 2 follows. Part 3 follows by symmetry.

We can derive a less expensive way to evaluate the formula for $\mathcal{B}_b^B(j)$ as follows. For $j = 0$, we have $\mathcal{B}_b^B(0) = \frac{b}{B}$. If $j > 0$, then

$$\begin{aligned}
\mathcal{B}_b^B(j) &= \frac{\binom{B-j-1}{b-1}}{\binom{B}{b}} \\
&= \frac{\frac{(B-j-1)!}{(b-1)!((B-j-1)-(b-1))!}}{\frac{B!}{b!(B-b)!}} \\
&= \frac{(B-j-1)!\ b!(B-b)!}{(b-1)!((B-j-1)-(b-1))!\ B!} \\
&= b\frac{(B-j-1)!\ (B-b)!}{((B-j-1)-(b-1))!\ B!} \\
&= b\frac{(B-j-1)!\ (B-b)!}{(B-j-b)!\ B!} \\
&= \frac{b}{B-j}\frac{(B-j)!\ (B-b)!}{(B-b-j)!\ B!} \\
&= \frac{b}{B-j}\prod_{i=0}^{j-1}(1-\frac{b}{B-i})
\end{aligned}$$

This formula is useful when $\mathcal{B}_b^B(j)$ occurs in sums over j because we can compute the product incrementally.

Corollary 4.16.10 *Using the terminology of Theorem 4.16.9, the expected value for the number of zeros*

1. *before the first one,*

2. *between two successive ones, and*

3. *after the last one*

is

$$\overline{\mathcal{B}}_b^B = \sum_{j=0}^{B-b} j\mathcal{B}_b^B(j) = \frac{B-b}{b+1} \tag{4.21}$$

Let us calculate:

$$\begin{aligned}
\sum_{j=0}^{B-b} j\binom{B-j-1}{b-1} &= \sum_{j=0}^{B-b}(B-(B-j))\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{B-j-1}{b-1} - \sum_{j=0}^{B-b}(B-j)\binom{B-j-1}{b-1} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{b-1} - b\sum_{j=0}^{B-b}\binom{B-j}{b} \\
&= B\sum_{j=0}^{B-b}\binom{b-1+j}{j} - b\sum_{j=0}^{B-b}\binom{b+j}{b} \\
&= B\binom{(b-1)+(B-b)+1}{(b-1)+1} - b\binom{b+(B-b)+1}{b+1} \\
&= B\binom{B}{b} - b\binom{B+1}{b+1} \\
&= (B-b\frac{B+1}{b+1})\binom{B}{b}
\end{aligned}$$

With

$$\begin{aligned}
B-b\frac{B+1}{b+1} &= \frac{B(b+1)-(Bb+b)}{b+1} \\
&= \frac{B-b}{b+1}
\end{aligned}$$

the claim follows.

Corollary 4.16.11 *Using the terminology of Theorem 4.16.9, the expected total number of bits from the first bit to the last one, both included, is*

$$\overline{\mathcal{B}}_{tot}(B,b) = \frac{Bb+b}{b+1} \tag{4.22}$$

To see this, we subtract from B the average expected number of zeros between the last one and the last bit:

$$\begin{aligned}
B-\frac{B-b}{b+1} &= \frac{B(b+1)}{b+1} - \frac{B-b}{b+1} \\
&= \frac{Bb+B-B+b}{b+1} \\
&= \frac{Bb+b}{b+1}
\end{aligned}$$

An early approximation of this formula was discovered by Kollias [494].

Corollary 4.16.12 *Using the terminology of Theorem 4.16.9, the number of bits from the first one and the last one, both included, is*

$$\overline{\mathcal{B}}_{\textit{1-span}}(B,b) = \frac{Bb - B + 2b}{b+1} \tag{4.23}$$

We have two possibilities to argue here. The first subtracts from B the number of zeros at the beginning and the end:

$$\begin{aligned}\overline{\mathcal{B}}_{\text{1-span}}(B,b) &= B - 2\frac{B-b}{b+1} \\ &= \frac{Bb + B - 2B + 2b}{b+1} \\ &= \frac{Bb - B + 2b}{b+1}\end{aligned}$$

The other possibility is to add the number of zeros between the first and the last one and the number of ones:

$$\begin{aligned}\overline{\mathcal{B}}_{\text{1-span}}(B,b) &= (b-1)\overline{\mathcal{B}}_b^B + b \\ &= (b-1)\frac{B-b}{b+1} + \frac{b(b+1}{b+1} \\ &= \frac{Bb - b^2 - B + b + b^2 + b}{b+1} \\ &= \frac{Bb - B + 2b}{b+1}\end{aligned}$$

The number of bits from the first bit to the last one including both . . . The distance between the first and the last one . . . EX or Cor?

Let us have a look at some possible applications of these formulas. If we look up one record in an array of B records and we search sequentially, how many array entries do we have to examine on average if the search is successful?

In [571] we find these formulas used for the following scenario. Let a file consist of B consecutive cylinders. We search for k different keys, all of which occur in the file. These k keys are distributed over b different cylinders. Of course, we can stop as soon as we have found the last key. What is the expected total distance the disk head has to travel if it is placed on the first cylinder of the file at the beginning of the search?

Another interpretation of these formulas can be found in [419, 572]. Assume we have an array consisting of B different entries. We sequentially go through all entries of the array until we have found all the records for b different keys. We assume that the B entries in the array and the b keys are sorted. Further, all b keys occur in the array. On the average, how many comparisons do we need to find all keys?

Vector of Buckets

A more general scenario is as follows. Consider a sequence of m buckets containing n_i items each. Yao [921] developed the following theorem.

Theorem 4.16.13 (Yao) *Consider a sequence of m buckets. For $1 \leq i \leq m$, let n_i be the number of items in a bucket i. Then there is a total of $N = \sum_{i=1}^{m} n_i$ items. Let*

$t_i = \sum_{l=0}^{i} n_i$ be the number of items in the first i buckets. If the buckets are searched sequentially, then the number of buckets that have to be examined until k distinct items have been found is

$$\mathcal{C}_{n_i}^{N,m}(k,j) = \frac{\binom{t_j}{k} - \binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{4.24}$$

Thus, the expected number of buckets that need to be examined in order to retrieve k distinct items is

$$\overline{\mathcal{C}}_{n_i}^{N,m}(k) = \sum_{j=1}^{m} j\mathcal{C}_{n_i}^{N,m}(k,j) = m - \frac{\sum_{j=1}^{m}\binom{t_{j-1}}{k}}{\binom{N}{k}} \tag{4.25}$$

Applications of this formula can be found in [168, 175, 571, 573, 845]. Manolopoulos and Kollias describe the analogue for the replacement model [571].

Lang, Driscoll, and Jou discovered a general theorem which allows us to estimate the expected number of block accesses for sequential search.

Theorem 4.16.14 (Lang/Driscoll/Jou) *Consider a sequence of N items. For a batched search of k items, the expected number of accessed items is*

$$A(N,k) = N - \sum_{i=1}^{N-1} \mathit{Prob}[Y \leq i] \tag{4.26}$$

where Y is a random variable for the last item in the sequence that occurs among the k items searched.

?

proof?

With the help of this theorem, it is quite easy to derive many average sequential accesses for different models.

Cor or EX?

4.16.3 Pointers into the Literature

Segments containing records can be organized differently. Records can be placed randomly in the segment, they can be ordered according to some key, or the segment is organized as a tree. Accordingly, the segment is called random, sequential, or tree-structure. From a segment, records are to be retrieved for a given bag of k keys. The general question then is: how many pages do we have to access? The answer depends on whether we assume the replacement or non-replacement model. Six cases occur. For sequential and tree-structured segments, it also makes sense to distinguish between successful, partially (un-) successful, and (totally) unsuccessfull searches. These notions capture the different possibilities where for all, some, none of the k keys records are found. The following table provides some entry points into the literature. It is roughly organized around the above categories. (Remember that we discussed the random file organization at length in Section 4.16.1.)

	non-replacement	replacement
random	[164, 168, 557, 659, 889, 920]	[111, 168, 642, 659]
sequential	[59, 168, 510, 573, 642, 641, 779, 921]	[168, 510, 573, 779]
tree-structured	[510, 509, 573, 641, 664]	[510, 509, 573, 779]

4.17 Disk Drive Costs for N Uniform Accesses

The goal of this section is to derive estimates for the costs (time) for retrieving N cache-missed sectors of a segment S from disk. We assume that the N sectors are read in their physical order on disk. This can be enforced by the DBMS, by the operating system's disk scheduling policy (SCAN policy), or by the disk drive controller.

Remembering the description of disk drives, the total costs can be described as

$$\mathcal{C}_{\text{disk}} = \mathcal{C}_{\text{cmd}} + \mathcal{C}_{\text{seek}} + \mathcal{C}_{\text{settle}} + \mathcal{C}_{\text{rot}} + \mathcal{C}_{\text{headswitch}} \tag{4.27}$$

For brevity, we omitted the parameter N and the parameters describing the segment and the disk drive on which the segment resides. Subsequently, we devote a (sometimes tiny) section to each summand. Before that, we have to calculate the number of qualifying cylinders, tracks, and sectors. These numbers will be used later on.

4.17.1 Number of Qualifying Cylinders, Tracks, and Sectors

If N sectors are to be retrieved, we have to find the number of cylinders qualifying in an extent i. Let S_{sec} denote the total number of sectors our segment contains and $S_{\text{cpe}}(i) = L_i - F_i + 1$ be the number of cylinders of the extent. If the N sectors we want to retrieve are uniformly distributed among the S_{sec} sectors of the segment, the number of cylinders that qualifies in (F_i, L_i, z_i) is $S_{\text{cpe}}(i)$ times 1 minus the probability that a cylinder does not qualify. The probability that a cylinder does not qualify can be computed by deviding the total number of possibilities to chose the N sectors from sectors outside the cylinder by the total number of possibilities to chose N sectors from all S_{sec} sectors of the segment. Hence, the number of qualifying cylinders in the considered extent is:

$$Q_c(i) = S_{\text{cpe}}(i)\mathcal{Y}^{S_{\text{sec}}}_{D_{\text{Zspc}}(i)}(N) = S_{\text{cpe}}(i)(1 - \frac{\binom{S_{\text{sec}} - D_{\text{Zspc}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \tag{4.28}$$

We could also have used Theorem 4.16.13.

Let us also calculate the number of qualifying tracks in a partion i. It can be calculated by $S_{\text{cpe}}(i)D_{\text{tpc}}(1 - \text{Prob}(\texttt{a track does not qualify}))$. The probability that a track does not qualify can be computed by dividing the number of ways to pick N sectors from sectors not belonging to a track divided by the number of possible ways to pick N sectors from all sectors.

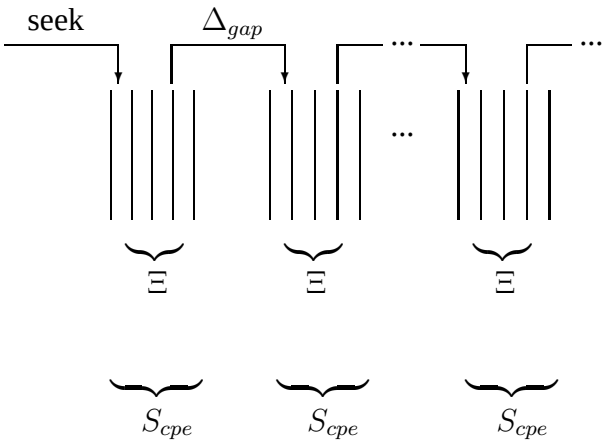
$$Q_t(i) = S_{\text{cpe}}(i)D_{\text{tpc}}\mathcal{Y}^{S_{\text{sec}}}_{D_{\text{Zspt}}(i)}(N) = S_{\text{cpe}}(i)D_{\text{tpc}}(1 - \frac{\binom{S_{\text{sec}} - D_{\text{Zspt}}(i)}{N}}{\binom{S_{\text{sec}}}{N}}) \tag{4.29}$$

Just for fun, we calculate the number of qualifying sectors of an extent in zone i. It can be approximated by

$$Q_s(i) = S_{\text{cpe}}(i)D_{\text{Zspc}}(i)\frac{N}{S_{\text{sec}}} \tag{4.30}$$

Since all $S_{\text{cpe}}(i)$ cylinders are in the same zone, they have the same number of sectors per track, and we could also use Waters/Yao to approximate the number of qualifying cylinders by

$$Q_c(i) = \overline{\mathcal{Y}}^{S_{\text{cpe}}(i)D_{\text{Zspc}}(S_{\text{zone}}(i)), S_{\text{cpe}}(i)}_{D_{\text{Zspc}}(S_{\text{zone}}(i))}(Q_s(i)) \tag{4.31}$$



The upper path illustrates $\mathcal{C}_{\text{seekgap}}$, the lower braces indicate those parts for which $\mathcal{C}_{\text{seekext}}$ is responsible.

Figure 4.10: Illustration of seek cost estimate

This is a good approximation, as long as $Q_s(i)$ is not too small (e.g. > 4).

4.17.2 Command Costs

The command costs $\mathcal{C}_{\text{cmd}}$ are easy to compute. Every read of a sector requires the execution of a command. Hence

$$\mathcal{C}_{\text{cmd}} = ND_{\text{cmd}}$$

estimates the total command costs.

4.17.3 Seek Costs

We give different alternative possibilities to estimate seek costs. We start with an upper bound by exploring Theorem 4.1.1. The first cylinder we have to visit requires a random seek with cost D_{avgseek}. (Well this does not really give us an upper bound. For a true upper bound we should use $D_{\text{seek}}(D_{\text{cyl}} - 1)$.) After that, we have to visit the remaining $Q_c(i) - 1$ qualifying cylinders. The segment spans a total of $S_{\text{last}}(S_{\text{ext}}) - S_{\text{first}}(1) + 1$ cylinders. Let us assume that the first qualifying cylinder is the first cylinder and the last qualifying cylinder is the last cylinder of the segment. Then applying Theorem 4.1.1 gives us the upper bound

$$\mathcal{C}_{\text{seek}}(i) \leq (Q_c(i) - 1)D_{\text{seek}}(\frac{S_{\text{last}}(S_{\text{ext}}) - S_{\text{first}}(1) + 1}{Q_c(i) - 1})$$

after we have found the first qualifying cylinder.

We can be a little more precise by splitting the seek costs into two components. The first component $\mathcal{C}_{\text{seekgap}}$ expresses the costs of finding the first qualifying cylinder and jumping from the last qualifying cylinder of extent i to the first qualifying cylinder

of extent $i+1$. The second component $\mathcal{C}_{\text{seekext}}(i)$ calculates the seek costs within an extent i. Figure 4.10 illustrates the situation. The total seek costs then are

$$\mathcal{C}_{\text{seek}}(i) = \mathcal{C}_{\text{seekgap}} + \sum_{i=1}^{S_{\text{ext}}} \mathcal{C}_{\text{seekext}}(i)$$

Since there is no estimate in the literature for $\mathcal{C}_{\text{seekgap}}$, we have to calculate it ourselves. After we have done so, we present several alternatives to calculate $\mathcal{C}_{\text{seekext}}(i)$.

The average seek cost for reaching the first qualifying cylinder is D_{avgseek}. How far are we now within the first extent? We use Corollary 4.16.10 to derive that the number of non-qualifying cylinders preceding the first qualifying one in some extent i is

$$\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)} = \frac{S_{\text{cpe}}(i) - Q_c(i)}{Q_c(i)+1}.$$

The same is found for the number of non-qualifying cylinders following the last qualifying cylinder. Hence, for every gap between the last and the first qualifying cylinder of two extents i and $i+1$, the disk arm has to travel the distance

$$\Delta_{\text{gap}}(i) := \overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)} + S_{\text{first}}(i+1) - S_{\text{last}}(i) - 1 + \overline{\mathcal{B}}^{S_{\text{cpe}}(i+1)}_{Q_c(i+1)}$$

Using this, we get

$$\mathcal{C}_{\text{seekgap}} = D_{\text{avgseek}} + \sum_{i=1}^{S_{\text{ext}}-1} D_{\text{seek}}(\Delta_{\text{gap}}(i))$$

Let us turn to $\mathcal{C}_{\text{seekext}}(i)$. We first need the number of cylinders between the first and the last qualifying cylinder, both included, in extent i. It can be calculated using Corollary 4.16.12:

$$\Xi_{\text{ext}}(i) = \overline{\mathcal{B}}_{\text{1-span}}(S_{\text{cpe}}(i), Q_c(i))$$

Hence, $\Xi(i)$ is the minimal span of an extent that contains all qualifying cylinders.

Using $\Xi(i)$ and Theorem 4.1.1, we can derive an upper bound for $\mathcal{C}_{\text{seekext}}(i)$:

$$\mathcal{C}_{\text{seekext}}(i) \leq (Q_c(i)-1) D_{\text{seek}}\left(\frac{\Xi(i)}{Q_c(i)-1}\right) \tag{4.32}$$

Alternatively, we could formulate this as

$$\mathcal{C}_{\text{seekext}}(i) \leq (Q_c(i)-1) D_{\text{seek}}\left(\overline{\mathcal{B}}^{S_{\text{cpe}}(i)}_{Q_c(i)}\right) \tag{4.33}$$

by applying Corollary 4.16.10.

A seemingly more precise estimate for the expected seek cost within the qualifying cylinders of an extent is derived by using Theorem 4.16.9:

$$\mathcal{C}_{\text{seekext}}(i) = Q_c(i) \sum_{j=0}^{S_{\text{cpe}}(i)-Q_c(i)} D_{\text{seek}}(j+1) \mathcal{B}^{S_{\text{cpe}}(i)}_{Q_c(i)}(j) \tag{4.34}$$

There are many more estimates for seek times. Older ones rely on a linear disk model but also consider different disk scan policies. A good entry point is the work by Theorey and Pinkerton [838, 839].

4.17.4 Settle Costs

The average settle cost is easy to calculate. For every qualifying cylinder, one head settlement takes place:

$$\mathcal{C}_{\text{settle}}(i) = Q_c(i) D_{\text{rdsettle}} \tag{4.35}$$

4.17.5 Rotational Delay Costs

Let us turn to the rotational delay. For some given track in zone i, we want to read the $Q_t(i)$ qualifying sectors contained in it. On average, we would expect that the read head is ready to start reading in the middle of some sector of a track. If so, we have to wait for $\frac{1}{2} D_{\text{Zscan}}(S_{\text{zone}}(i))$ before the first whole sector occurs under the read head. However, due to track and cylinder skew, this event does not occur after a head switch or a cylinder switch. Instead of being overly precise here, we ignore this half sector pass by time and assume that we are always at the beginning of a sector. This is also justified by the fact that we model the head switch time explicitly.

EX

Assume that the head is ready to read at the beginning of some sector of some track. Then, in front of us is a — cyclic, which does not matter — bitvector of qualifying and non-qualifying sectors. We can use Corollary 4.16.11 to estimate the total number of qualifying and non-qualifying sectors that have to pass under the head until all qualifying sectors have been seen. The total rotational delay for the tracks of zone i is

$$\mathcal{C}_{\text{rot}}(i) = Q_t(i)\ \ D_{\text{Zscan}}(S_{\text{zone}}(i))\ \ \overline{\mathcal{B}}_{\text{tot}}(D_{\text{Zspt}}(S_{\text{zone}}(i)), Q_{\text{spt}}(i)) \tag{4.36}$$

where $Q_{\text{spt}}(i) = \overline{\mathcal{W}}_1^{S_{\text{sec}}, D_{\text{Zspt}}(S_{\text{zone}}(i))}(N) = D_{\text{Zspt}}(S_{\text{zone}}(i)) \frac{N}{S_{\text{sec}}}$ is the expected number of qualifying sectors per track in extent i. In case $Q_{\text{spt}}(i) < 1$, we set $Q_{\text{spt}}(i) := 1$.

A more precise model is derived as follows. We sum up for all j the product of (1) the probability that j sectors in a track qualify and (2) the average number of sectors that have to be read if j sectors qualify. This gives us the number of sectors that have to pass the head in order to read all qualifying sectors. We only need to multiply this number by the time to scan a single sector and the number of qualifying tracks. We can estimate (1) using Theorem 4.16.8. For (2) we again use Corollary 4.16.11.

$$\begin{aligned} \mathcal{C}_{\text{rot}}(i) \quad = \quad & S_{\text{cpe}}(i)\ \ D_{\text{tpc}}\ \ D_{\text{Zscan}}(S_{\text{zone}}(i)) \\ & \cdot \sum_{j=1}^{\min(N, D_{\text{Zspt}}(S_{\text{zone}}(i)))} \mathcal{X}_{D_{\text{Zspt}}(S_{\text{zone}}(i))}^{S_{\text{sec}}}(N, j)\ \ \overline{\mathcal{B}}_{\text{tot}}(D_{\text{Zspt}}(S_{\text{zone}}(i)), j) \end{aligned} \tag{4.37}$$

Another approach is taken by Triantafillou, Christodoulakis, and Georgiadis [845]. They split the total rotational delay into two components. The first component ($\mathcal{C}_{\text{rotpass}}$) measures the time needed to skip unqualifying sectors and the second ($\mathcal{C}_{\text{rotread}}$) that for (scanning and transferring) the qualifying sectors to the host.

Let us deal with the first component. Assume that j sectors of a track in extent i qualify. The expected position on a track where the head is ready to read is the middle between two qualifying sectors. Since the expected number of sectors between two qualifying sectors is $D_{\text{Zspt}}(S_{\text{zone}}(i))/j$, the expected number of sectors scanned before the first qualifying sector comes under the head is $\frac{D_{\text{Zspt}}(S_{\text{zone}}(i))}{2j}$. The expected

positions of j qualifying sectors on the same track is such that the number of non-qualifying sectors between two successively qualifying sectors is the same. Hence, after having read a qualifying sector, $\frac{D_{\text{Zspt}}(S_{\text{zone}}(i))}{j}$ unqualifying sectors must pass by until the next qualifying sector shows up. The total number of unqualifying sectors to be passed if j sectors qualify in a track of zone i is

$$N_s(j,i) = \frac{D_{\text{Zspt}}(S_{\text{zone}}(i))}{2j} + (j-1)\frac{D_{\text{Zspt}}(S_{\text{zone}}(i)) - j}{j} \tag{4.38}$$

Using again Theorem 4.16.8, the expected rotational delay for the unqualifying sectors then is

$$\begin{aligned} \mathcal{C}_{\text{rotpass}}(i) &= S_{\text{cpe}}(i)\ D_{\text{tpc}}\ D_{\text{Zscan}}(S_{\text{zone}}(i)) \\ &\quad \cdot \sum_{j=1}^{\min(N, D_{\text{Zspt}}(S_{\text{zone}}(i)))} \mathcal{X}^{S_{\text{sec}}}_{D_{\text{Zspt}}(S_{\text{zone}}(i))}(N,j) N_s(j,i) \end{aligned} \tag{4.39}$$

We have to sum up this number for all extents and then add the time needed to scan the N sectors. Hence

$$\mathcal{C}_{\text{rot}} = \sum_{i=1}^{S_{\text{ext}}} \mathcal{C}_{\text{rotpass}}(i) + \mathcal{C}_{\text{rotread}}(i)$$

where the total transfer cost for the qualifying sectors of an extent can be estimated as

$$\mathcal{C}_{\text{rotread}}(i) = Q_s(i)\ D_{\text{Zscan}}(S_{\text{zone}}(i))$$

4.17.6 Head Switch Costs

The average head switch cost is equal to the average number of head switches that occur times the average head switch cost. The average number of head switch is equal to the number of tracks that qualify minus the number of cylinders that qualify since a head switch does not occur for the first track of each cylinder. Summarizing

$$\mathcal{C}_{\text{headswitch}} = \sum_{i=1}^{S_{\text{ext}}} (Q_t(i) - Q_c(i))\ D_{\text{hdswitch}} \tag{4.40}$$

where Q_t is the average number of tracks qualifying in an extent.

4.17.7 Discussion

The disk drive cost model derived depends on many parameters. The first bunch of parameters concerns the disk drive itself. These parameters can (and must be) extracted from disk drives by using (micro-) benchmarking techniques [295, 834, 588, 738]. The second bunch of parameters concerns the layout of a segment on disk. The database system is responsible for providing these parameters. The closer it is to the disk, the easier these parameters are extracted. Building a runtime system atop the operating system's file system is obviously a bad idea from the cost model perspective. If instead the storage manager of the runtime system implements cylinder aligned extents (or at least track aligned extents) using a raw I/O interface, the cost model will be easier to

develop and much more precise. Again, providing reliable cost models is one of the most important tasks of the runtime system.

We have neglected many problems in our disk access model: partially filled cylinders, pages larger than a block, disk drive's cache, remapping of bad blocks, non-uniformly distributed accesses, clusteredness, and so on. Whereas the first two items are easy to fix, the rest is not so easy. In general, database systems ignore the disk drive cache. The justifying argument is that the database buffer is much larger than the disk drive's cache and, hence, it is very unlikely that we read a page that is not in the database buffer but in the disk cache. However, this argument falls short for non-random accesses. Nevertheless, we will ignore the issue in this book. The interested reader is referred to Shriver's thesis for disk cache modeling [780].

Remapping of bad sectors to other sectors really prevents the development of a precise cost model for disk accesses. Modelling disk drives becomes already a nightmare since a nice partitioning of the disk into zones is no longer possible since some sectors, tracks and even cylinders are reserved for the remapping. So even if no remapping takes place (which is very unlikely), having homogeneous zones of hundreds of cylinders is a dream that will never come true. The result is that we do not have dozens of homogeneous zones but hundreds (if not thousands) of zones of medium homogeneity. These should be reduced to a model of dozens of homogeneous zones such
EX that the error does not become too large. The remaining issues will be discussed later in the book.

There is even more to say about our cost model. A very practical issue arises if the number of qualifying cylinders is small. Then for some extent i, the expected number of qualifying cylinders could be $Q_c(i) = 0.38$. For some of our formulas this is a big problem. As a consequence, special cases for small N, small Q_c, small Q_t have to be
EX developed and implemented.

Another issue is the performance of the cost model itself. The query compiler might evaluate the cost model's formulas thousands or millions of times. Hence, they should be fast to evaluate.

So far, we can adequately model the costs of N disk accesses. Some questions remain. For example, how do we derive the number N of pages we have to access? Do we really need to fetch all N pages from disk or will we find some of them in the buffer? If yes, how many? Further, CPU costs are also an important issue. Deriving a cost model for CPU costs is even more tedious than modelling disk drive costs. The only choice available is to benchmark all parts of a system and then derive a cost model using the extracted parameters. To give examples of parameters to be extracted: we need the CPU costs for accessing a page present in the buffer, for accessing a page absent in the buffer, for a next call of an algebraic operator, for executing an integer addition, and so on. Again, this cannot be done without tools [45, 237, 403, 452, 658].

The bottom line is that a cost model does not have to be accurate, but must lead to correct decisions. In that sense, it must be accurate at the break even points between plan alternatives. Let us illustrate this point by means of our motivating example. If we know that the index returns a single tuple, it is quite likely that the sequential scan is much more expensive. The same might be true for 2, 3, 4, and 5 tuples. Hence, an accurate model for small N is not really necessary. However, as we come close to the costs of a sequential scan, both the cost model for the sequential scan and the one for the index-based access must be correct since the product of their errors is

the factor a bad choice is off the best choice. This is a crucial point, since it is easy to underestimate sequential access costs by a factor of 2-3 and overestimate random access cost by a factor of 2-5.

4.18 Concluding Remarks

Learned:

Open Cost: I/O costs: non-uniform stuff, CPU costs: nothing done
Wrong cardinality estimates: Open, leads to dynamic qo

4.19 Bibliography

ToDo:

- CPU Costs for B-tree search within inner and leaf pages [507]
- Index/Relations: only joins between building blocks [707]
- RDB/V1: predicate push down (views), 2 phase optimization (local: traditional, global: sharing of tables), five categories for predicates, nested loops evaluation for nested correlated subqueries, use of transitivity of equality, conjunctive normal form, use of min/max value of join column to reduce join cardinality by adding another selection to the other relation (min(a) $<=$ b $<=$ max(a)) for join predicate a=b.
- K accesses to unique index: how many page faults if buffer has size b? [720]
- buffer mgmt: [247]
- buffer mgmt: [778]
- buffer mgmt: [511]
- buffer mgmt: [721, 722]
- buffer mgmt: [99]
- buffer mgmt: [166]
- structured, semi-structured, unstructured data: [328] cited in Dono76
- B-trees and their improvements [209]
- Vertical partitioning: [251, 574, 52]
- Horizontal and vertical partitioning: [120]
- set oriented disk access to large complex objects [882, 881], assembly operator: [471],
- large objects: [82, 112, 526]

Part II

Foundations

Chapter 5

Logic, Null, and Boolean Expressions

5.1 Two-valued logic

The Boolean algebra with its operations *not* ($\neg$), *and* ($\wedge$), and *or* ($\vee$) is well-known. The truth tables for these operations is given in Figure 5.1.

5.2 NULL values and two valued logic

5.3 Three valued logic

5.4 Simplifying Boolean Expressions

5.5 Optimizing Boolean Expressions

5.6 Bibliography

NULL-values: [715, 531, 716, 532]

$\neg$		
	true	*false*
	false	*true*

$\vee$		
	true	*false*
true	*true*	*true*
false	*true*	*false*

$\wedge$		
	true	*false*
true	*true*	*false*
false	*false*	*false*

Figure 5.1: Truth tables for two-valued logic

$$\begin{array}{lcl} \neg \mathit{true} & \Longrightarrow & \mathit{false} \\ \neg \mathit{false} & \Longrightarrow & \mathit{true} \end{array}$$

$$\begin{array}{lcl} p \vee p & \Longrightarrow & p \\ p \vee \neg p & \Longrightarrow & \mathit{true} \\ p_1 \vee (p_1 \wedge p_2) & \Longrightarrow & p_1 \\ p \vee \mathit{false} & \Longrightarrow & p \\ p \vee \mathit{true} & \Longrightarrow & \mathit{true} \end{array} \qquad \begin{array}{lcl} p \wedge p & \Longrightarrow & p \\ p \wedge \neg p & \Longrightarrow & \mathit{false} \\ p_1 \wedge (p_1 \vee p_2) & \Longrightarrow & p_1 \\ p \wedge \mathit{true} & \Longrightarrow & p \\ p \vee \mathit{false} & \Longrightarrow & \mathit{false} \end{array}$$

Figure 5.2: Simplification Rules

Commutativity

$$p_1 \vee p_2 \iff p_2 \vee p_1 \qquad p_1 \wedge p_2 \iff p_2 \wedge p_1$$
$$\exists e_1\, \exists e_2\, p \iff \exists e_2\, \exists e_1\, p \qquad \forall e_1\, \forall e_2\, p \iff \forall e_2\, \forall e_1\, p$$

Associativity

$$(p_1 \vee p_2) \vee p_3 \iff p_1 \vee (p_2 \vee p_3) \qquad (p_1 \wedge p_2) \wedge p_3 \iff p_1 \wedge (p_2 \wedge p_3)$$

Distributivity

$$p_1 \vee (p_2 \wedge p_3) \iff (p_1 \vee p_2) \wedge (p_1 \vee p_3) \qquad p_1 \wedge (p_2 \vee p_3) \iff (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$$
$$\exists e\, (p_1 \vee p_2) \iff (\exists e\, p_1) \vee (\exists\, p_2) \qquad \forall e\, (p_1 \wedge p_2) \iff (\forall e\, p_1) \wedge (\forall e\, p_2)$$

Idempotency

$$p \vee p \iff p \qquad p \wedge p \iff p$$
$$p \vee \neg p \iff \textit{true} \qquad p \wedge \neg p \iff \textit{false}$$
$$p_1 \vee (p_1 \wedge p_2) \iff p_1 \qquad p_1 \wedge (p_1 \vee p_2) \iff p_1$$
$$p \vee \textit{false} \iff p \qquad p \wedge \textit{true} \iff p$$
$$p \vee \textit{true} \iff \textit{true} \qquad p \wedge \textit{false} \iff \textit{false}$$

De Morgan

$$\neg(p_1 \vee p_2) \iff \neg(p_1) \wedge \neg(p_2) \qquad \neg(p_1 \wedge p_2) \iff \neg(p_1) \vee \neg(p_2)$$

Negation of Quantifiers

$$\neg(\forall e\, p) \iff \exists e(\neg p) \qquad \neg(\exists e\, p) \iff \forall e\, (\neg p)$$

Elimination of Negation

$$\neg(\neg(p)) \iff p \qquad \neg t_1 \theta t_2 \implies t_1 \overline{\theta} t_2$$

Conditioned Distributivity $(\mathcal{F}(p_1) \cap \mathcal{A}(e) = \emptyset)$

$$p_1 \vee (\forall e\, p_2) \iff \forall e\, (p_1 \vee p_2) \qquad p_1 \wedge (\exists e\, p_2) \iff \exists e\, (p_1 \wedge p_2)$$
$$p_1 \vee (\exists e\, p_2) \iff \begin{cases} \exists e(p_1 \vee p_2) & \textit{if } e \neq \{\} \\ p_1 & \textit{if } e = \{\} \end{cases}$$
$$p_1 \wedge (\forall e\, p_2) \iff \begin{cases} \forall e(p_1 \wedge p_2) & \textit{if } e \neq \{\} \\ p_1 & \textit{if } e = \{\} \end{cases}$$

Figure 5.3: Laws for two-valued logic

$\neg$	
true	*false*
false	*true*
$\bot$	$\bot$

$\vee$	*true*	*false*	$\bot$
true	*true*	*true*	*true*
false	*true*	*false*	$\bot$
$\bot$	*true*	$\bot$	$\bot$

$\wedge$	*true*	*false*	$\bot$
true	*true*	*false*	$\bot$
false	*false*	*false*	*false*
$\bot$	$\bot$	*false*	$\bot$

Figure 5.4: Truth tables for three-valued logic

Chapter 6

An Algebra for Sets, Bags, and Sequences

6.1 Sets, Bags, and Sequences

6.1.1 Sets

Sets and their laws (see Figure 6.1) should be well-known. A set contains elements drawn from some domain $\mathcal{D}$. In our case, the domain will often be tuples and we only consider finite sets.

The set operations we are interested in are union ($\cup$), intersection ($\cap$), and set difference ($\setminus$). If the domain consists of tuples, we assume that both arguments have the same schema. That is, the attributes and their domains are the same in both arguments. Otherwise, the expression is not well-typed. In any case, set union and intersection are commutative and associative. Set difference is neither of them. Expressions containing

$$
\begin{array}{lcll}
X \cup \emptyset & = & X & \\
X \cup X & = & X & \\
X \cup Y & = & Y \cup X & (commutativity) \\
(X \cup Y) \cup Z & = & X \cup (Y \cup Z) & (associativity) \\
X \cap \emptyset & = & \emptyset & \\
X \cap x & = & X & \\
X \cap Y & = & y \cap X & (commutativity) \\
(X \cap Y) \cap Z & = & X \cap (Y \cap z) & (associativity) \\
X \setminus \emptyset & = & X & \\
\emptyset \setminus X & = & \emptyset & \\
X \setminus Y & \neq & Y \setminus X & (***wrong***) \\
(X \setminus Y) \setminus Z & \neq & X \setminus (Y \setminus Z) & (***wrong***) \\
X \cup (Y \cap Z) & = & (X \cap Y) \cup (X \cap Z) & (distributivity) \\
X \cap (Y \cup Z) & = & (X \cup Y) \cap (X \cup Z) & (distributivity) \\
(X \cup Y) \setminus Z & = & (X \setminus Z) \cup (Y \setminus Z) & (distributivity)
\end{array}
$$

Figure 6.1: Laws for Set Operations

the empty set can simplified. Further, some distributivity laws hold.

As we have seen in Chapter 2, algebraic equivalences that reorder algebraic operators form the fundamental core for query optimization. One could discuss the reorderability of each pair of operators resulting in n^2 investigations if the number of operators in the algebra is n. In order to simplify this, we introduce a general argument covering most of the cases. The observation will be that *set-linearity* of set operators implies their reorderability easily.

A unary function f from sets to sets is called *set-linear* (or *homomorph*), if and only if the following two conditions hold for all sets X and Y:

$$\begin{aligned} f(\emptyset) &= \emptyset \\ f(X \cup Y) &= f(X) \cup f(Y) \end{aligned}$$

An n-ary mapping from sets to a set is called *set-linear in its i-th argument*, if and only if for all sets $X_1, \ldots, X_n$ and X_i' the following conditions hold:

$$\begin{aligned} f(X_1, \ldots, X_{i-1}, \emptyset, X_{i+1}, \ldots, X_n) &= \emptyset \\ f(X_1, \ldots, X_{i-1}, X_i \cup X_i', X_{i+1}, \ldots, X_n) &= f(X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n) \\ &\quad \cup f(X_1, \ldots, X_{i-1}, X_i', X_{i+1}, \ldots, X_n) \end{aligned}$$

It is called *set-linear*, if it is linear in all its arguments. For a binary function or operator where we can distinguish between the left and the right argument, we call it left (right) set-linear if it is set-linear in its first (second) argument. Note that if an equivalence with linear mappings on both sides has to be proven, it suffices to proof it for disjoint singleton sets, i.e. sets with one element only.

Set intersecion is set-linear, set difference is left set-linear but not right set-linear and set union is neither left nor right set-linear.

A set of elements from a domain $\mathcal{D}$ can be seen a function from $\mathcal{D}$ to $\{0, 1\}$. For a given set S, this function is called the *characteristic function* of S. It can be defined as $\chi_S(s) = \begin{cases} 0 & \text{if } s \notin S \\ 1 & \text{if } s \in S \end{cases}$ Of course their is a bijection between characteristic functions and sets.

6.1.2 Duplicate Data: Bags

A *bag* or *multiset* is can contain every element more than once. It cannot contain an element less than zero times. A typical bag is $\{\{a, b, b\}\}$. Another example is $\{\{a, b\}\}$. The latter bag does not contain any duplicates. Hence, it could also be considered a set.

Hence, for a given bag B, the characteristic function for bags maps every element of a domain D to the set of non-negative integers $\mathbb{N}_0$. The characteristic function gives the number of occurrences of each element in the bag. Again, there is a bijection between bags and their characteristic functions. We will only consider finite bags.

The bag union $X \uplus Y$ of two bags is defined such that the number of occurrences of some element in the union is the sum of its occurrences in X and Y: The number of occurrences of some element in the bag intersection $X \mathbin{\dot\cap} Y$ is the minimum of the

$$
\begin{array}{lcll}
X \bar{\cup} \bar{\emptyset} & = & X & \\
X \bar{\cup} X & = & X & \\
X \bar{\cup} Y & = & Y \bar{\cup} X & (commutativity) \\
(X \bar{\cup} Y) \bar{\cup} Z & = & X \bar{\cup} (Y \bar{\cup} Z) & (associativity) \\
X \bar{\cap} \bar{\emptyset} & = & \bar{\emptyset} & \\
X \bar{\cap} x & = & X & \\
X \bar{\cap} Y & = & Y \bar{\cap} X & (commutativity) \\
(X \bar{\cap} Y) \bar{\cap} Z & = & X \bar{\cap} (Y \bar{\cap} Z) & (associativity) \\
X \bar{\setminus} \bar{\emptyset} & = & X & \\
\bar{\emptyset} \bar{\setminus} X & = & \bar{\emptyset} & \\
X \bar{\setminus} Y & \neq & Y \bar{\setminus} X & (***wrong***) \\
(X \bar{\setminus} Y) \bar{\setminus} Z & \neq & X \bar{\setminus} (Y \bar{\setminus} Z) & (***wrong***) \\
X \bar{\cup} (Y \bar{\cap} Z) & = & (X \bar{\cap} Y) \bar{\cup} (x \bar{\cap} Z) & (distributivity) \\
X \bar{\cap} (Y \bar{\cup} Z) & \neq & (X \bar{\cup} Y) \bar{\cap} (X \bar{\cup} Z) & (***wrong***) \\
(X \bar{\cup} Y) \bar{\setminus} Z & \neq & (X \bar{\setminus} Z) \bar{\cup} (Y \bar{\setminus} Z) & (***wrong***)
\end{array}
$$

Figure 6.2: Laws for Bag Operations

number its occurrences in X and Y. In the bag difference $X \bar{\setminus} Y$ the number of occurrences of some element is the difference ($\dot{-}$) of its occurrences in X and Y where $a \dot{-} b$ is defined as $\max(0, a - b)$. Using characteristic functions, we can define

$$
\begin{array}{lcl}
\chi_{X \bar{\cup} Y}(z) & = & \chi_X(z) + \chi_Y(z) \\
\chi_{X \bar{\cap} Y}(z) & = & \min(\chi_X(z), \chi_Y(z)) \\
\chi_{X \bar{\setminus} Y}(z) & = & \chi_X(z) \dot{-} \chi_Y(z)
\end{array}
$$

The laws for sets do not necessarily hold for bags (see Figure 6.2). We will have that bag union and bag intersection are both commutative and associative. Bag difference is neither of them. The only distributivity law that holds for bags is the one that distributes a union over an intersection.

Having every operation twice, once for bags and once for sets is quite inconvenient. Fortunately, for some operations we only need the one for bags. We can get rid of some set operations as follows. Every set can be seen as a bag whose the characteristic function never exceeds one. Let $\bar{I}(S)$ turn a set S into a bag with identical characteristic function. The partial function $\bar{I}^{-1}(B)$ return a bag into a set if the bag's characteristic function does not exceed one. Otherwise let $\bar{I}^{-1}$ be undefined. Let X and Y be two sets. For the intersection function we then have

$$\bar{I}^{-1}(\bar{I}(X) \bar{\cup} \bar{I}(Y)) = X \cup Y$$

That is, for any two sets X and Y bag union and set union are the same. This gives rise to the notion of *set-faithfulness*. We call a unary function on sets f *set-faithtul* if and only if

$$\bar{I}^{-1}(f(\bar{I}(X))) = f(X)$$

holds for all sets X. Analogously, binary functions g are *set-faithful* if and only if

$$\bar{I}^{-1}(g(\bar{I}(X), \bar{I}(Y))) = f(X, Y)$$

holds for all sets X and Y.

$\bar{\setminus}$ and $\bar{\cap}$ are set-faithful. Hence, we can (and often will) simply use $\setminus$ and $\cap$ to denote set or bag difference and intersection. However, $\bar{\cup}$ is not. Hence, we have to carefully distinguish between $\bar{\cup}$ and $\cup$.

To go from a bag to a set, we have to eliminate duplicates. Let us denote by Π^D the duplicate elimination operation. For a given bag B we then have $\chi_{\Pi^D(B)}(z) = \min(1, \chi_B(z))$.

Instead of working with sets and bags, we can work with bags only by identifying every set S with the bag $\bar{I}(S)$. We can annotate all bags with a property indicating whether it contains duplicates or not. If at some place a set is required and we cannot infer that the bag in that place is duplicate free, we can use Π^D as an enforcer of the set property. Note that for every set S we have $\Pi^D(S) = S$. Hence, Π^D does not do any harm except for the resources it takes. The reasoning whether a given expression produces duplicates or not will be very important, especially in the context of XPath/XQuery.

A unary function f from bags to bags is called *bag-linear* (or *homomorph*), if and only if the following two conditions hold for all bags X and Y:

$$\begin{aligned} f(\bar{\emptyset}) &= \bar{\emptyset} \\ f(X \bar{\cup} Y) &= f(X) \bar{\cup} f(Y) \end{aligned}$$

An n-ary mapping from bags to a bag is called *bag-linear in its i-th argument*, if and only if for all bags $X_1, \ldots, X_n$ and X_i' the following conditions hold:

$$\begin{aligned} f(X_1, \ldots, X_{i-1}, \bar{\emptyset}, X_{i+1}, \ldots, X_n) &= \bar{\emptyset} \\ f(X_1, \ldots, X_{i-1}, X_i \cup X_i', X_{i+1}, \ldots, X_n) &= f(X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n) \\ &\quad \bar{\cup} f(X_1, \ldots, X_{i-1}, X_i', X_{i+1}, \ldots, X_n) \end{aligned}$$

It is called *bag-linear*, if it is linear in all its arguments. For a binary function or operator where we can distinguish between the left and the right argument, we call it left (right) bag-linear if it is bag-linear in its first (second) argument. Note that if an equivalence with linear mappings on both sides has to be proven, it suffices to proof it for disjoint singleton bags, i.e. bags with one element only.

None of bag union, intersection, and difference are left or right bag-linear.

TODO

- $(X \bar{\cup} Y) \bar{\setminus} Z \neq (X \bar{\setminus} Z) \bar{\cup} (Y \bar{\setminus} Z)$
- $\mathcal{A}(()R) = \mathcal{A}(()S)$
  $(R \bar{\bowtie} S) \neq (R \bar{\cap} S)$

6.1.3 Ordered Data: Sequences

A sequence is ordered and may contain duplicates. An example sequence is $\langle a, b, b, c, b \rangle$. The length of the sequence is the number of elements it contains. For any sequence S, the length of the sequence is denoted by $|S|$. The above sequence has length five. The empty sequence (ϵ) contains zero elements and has length zero.

As we consider only finite sequences, a sequence of length $n \geq 0$ has a characteristic function χ from an interval $[0, n[$ to a domain D. Outside $[0, n[$, χ is undefined

($\perp$). Let S be a sequence. Then $\alpha(S)$ gives us the first element of the sequence, i.e. $\alpha(S) = \chi_S(0)$. For our example sequence, $\alpha(\langle a,b,b,c,b\rangle) = a$. The rest or tail of a sequence S of length n is denoted by $\tau(S)$ and contains all but the first element in the sequence. That is $\chi_{\tau(S)}(i) = \chi_S(i+1)$. For our example sequence, $\tau(\langle a,b,b,c,b\rangle) = \langle b,b,c,b\rangle$.

Concatenation of two sequences is denoted by $\oplus$. The characteristic function of the concatenation of two sequences S and T is $\chi_{S\oplus T}(i) = \begin{cases} \chi_S(i) & \text{if } i < |S| \\ \chi_T(i-|S|) & \text{if } i \geq |S| \end{cases}$ As an example, $\langle a,b,b,c,b\rangle \oplus \langle a,b,c\rangle = \langle a,b,b,c,b,a,b,c\rangle$.

We can easily go from a sequence to a bag by just forgetting the order. To convert a bag into a sequence, we typically have to apply a `Sort` operator. In reality however, bags are often represented as (ordered) streams, i.e. they are sequences. This is due to fact that most physical algebras are implemented using the iterator concept introduced in Section 4.6.

Analogously to set and bag linearity, we can introduce *sequence linearity* of unary and n-ary functions on sequences. In the definition, we only have to exchange the set (bag) union operator by concatentation:

A unary function f from sequences to sequences is called *sequence-linear*, if and only if the following two conditions hold for all sequences X and Y:

$$\begin{aligned} f(\epsilon) &= \epsilon \\ f(X \oplus Y) &= f(X) \oplus f(Y) \end{aligned}$$

An n-ary mapping from sequences to a sequence is called *sequence-linear in its i-th argument*, if and only if for all sequences $X_1, \ldots, X_n$ and X_i' the following conditions hold:

$$\begin{aligned} f(X_1,\ldots,X_{i-1},\epsilon,X_{i+1},\ldots,X_n) &= \epsilon \\ f(X_1,\ldots,X_{i-1},X_i \cup X_i',X_{i+1},\ldots,X_n) &= f(X_1,\ldots,X_{i-1},X_i,X_{i+1},\ldots,X_n) \\ &\quad \oplus f(X_1,\ldots,X_{i-1},X_i',X_{i+1},\ldots,X_n) \end{aligned}$$

It is called *sequence-linear*, if it is sequence-linear in all its arguments. For a binary function or operator where we can distinguish between the left and the right argument, we call it left (right) sequence-linear if it is sequence-linear in its first (second) argument. Note that if an equivalence with linear mappings on both sides has to be proven, it suffices to proof it for disjoint singleton sequences, i.e. sequences with one element only.

6.2 Algebra

ToDo Grouping Mapping and Commutativity Comment on $\Pi_A(R_1 \cap R_2) \not\equiv \Pi_A(R_1) \cap \Pi_A(R_2)$ $\Pi_A(R_1 \setminus R_2) \not\equiv \Pi_A(R_1) \setminus \Pi_A(R_2)$

This section summarizes a logical algebra that is sufficent to express plans for SQL, OQL and XPath/XQuery queries. The algebra is an extension of a union of algebras developed by many people [64, 66, 186, 187, 475, 476, 528, 790]. The most prominent features of the algebra are:

- All operators are polymorphic and can deal with (almost) any kind of complex arguments.
- The operators take arbitrary complex expressions as subscripts. This includes algebraic expressions.
- The algebra is redundant since some special cases of the operators can be implemented much more efficiently. These cases are often introduced as abbreviations.

The remainder of this subsection is organized by operator. The core of the algebra consists of the following operators that are all defined on sets, bags, and sequences. We have already seen the union, intersubsection, and difference operators. The others are selection (σ), projection (Π), map (χ), join($\bowtie$), semi-join ($\rtimes$), anti-join ($\triangleright$), left-outer join ($\overrightarrow{\bowtie}$), d-join ($< \cdot >$), map (χ), unnest (μ), and unary (Γ) and binary grouping (Γ). For every algebraic operator we have a subsection discussing it.

While some of the operators are simple extensions of standard operators, others are not widely spread. This forces us to reconsider existing algebraic equivalences as well as to discover new ones. The main question here is reorderability of operators since reorderability is fundamental to any optimization process. Hence, we will discuss reorderability results. Thereby, we will make use of the notion of linearity.

ToDo Since the join remains the most expensive operator in our algebra, subsubsection 6.2.11 discusses simplifications of expressions containing joins.

6.2.1 The Operators

The set operators as well as the selection operator and the different join operators (except for the d-join) are known from the relational context. As we will see, the only difference in their definition is that they are extended to express nested queries. For this, we allow the subscripts of these operators to contain full algebraic expressions. Further, to adjust them to the object-oriented context, they do not only deal with relations, but with sets, bags, and sequences of items, where items can be simple tuples, tuples where attributes are allowed to have complex values, and even non-tuple items (e.g. objects or document nodes). This means, that the attribute values are in no way restricted to atomic types but can carry also objects, sets, bags, sequences and so on.

The left-outer join additionally needs some tuning in order to exploit the possibility to have sets as attribute values. For this, it carries a superscript giving a default value for some attribute for those tuples in the left argument for which there is no matching tuple in the right argument. The d-join operation is used for performing a join between

two bulk valued items, where the second one is dependent on the first one. This operator can be used for unnesting nested queries and is in many cases equivalent to a join between two sets with a membership predicate [773]. In some cases (as we will see later on), it corresponds to an unnest operation. We introduced the d-join in order to cope with the values of types that do not have extensions (i.e. there exist no explicit set on which a join could be applied). The d-join is also useful for introducing index structures.

The map operator χ ([475]) is well-known from the functional programming language context. A special case of it, where it adds derived information in form of an added attribute with an according value (e.g. by object-base lookup or by method calls) to each tuple of a set has been proposed in [474, 475]. Later, this special case was given the name *materialization operator* [85].

The unnest operator is known from NF^2 [732, 714]. It will come in two different flavors allowing us to perform unnesting not only on nested relations but also on attributes whose value is a bulk of elements which are not tuples. The last operator—the *grouping* operator—generalizes the nest operator quite a bit. That is why we renamed it. In fact, there exist two grouping operators, one unary grouping operator and one binary grouping operator. The unary grouping operator groups one set of tuples according to a grouping condition. Further, it can apply an arbitrary expression to the newly formed group. This allows an efficient implementation by saving on intermediate results. The binary grouping operator adds a group to each element in the first argument set. This group is formed from the second argument. The grouping operator will exploit the fact that in the object-oriented context attributes can have set-valued attributes. As we will see, this is useful for both, unnesting nested queries and producing nested results. A variant of the binary grouping operator is also called *nest-join* [800]. Implementations of it are discussed in [131]. There, the binary grouping operator is called *MD-Join*.

6.2.2 Preliminaries

As already mentioned, our algebraic operators can deal not only with standard relations but are polymorphic in the general sense. In order to fix the domain of the operators we need some technical abbreviations and notations. Let us introduce these first.

Since our operators are polymorphic, we need variables for types. We use τ possibly with a subscript to denote types. To express that a certain expression is of type e, we write $e :: \tau$. Starting from concrete names for types and type variables, we can build type expressions the standard way by using type constructors to build tuple tupes $([\cdot])$, set types $\{\cdot\}$, bag types $\{\{\cdot\}\}$ and sequence types $< \cdot >$. Having two type expressions t_1 and t_2, we denote by $t_1 \leq t_2$, that t_1 is a subtype of t_2. It is important to note that this subtype relationship is not based on the sub-/superclass hierarchy found in most object-oriented models. Instead, it simply denotes substitutability. That is the type t_1 provides at least all the attributes and member functions that t_2 provides [110].

Most of our algebraic operators are tuned to work on bulks of tuples. The most important information here is the attributes provided. For this we introduce $\mathcal{A}(A)$. The function $\mathcal{A}$ is defined as follows. $\mathcal{A}(e) = \{a_1, \ldots, a_n\}$ if $e :: \{[a_1 : \tau_1, \ldots, a_n : \tau_n]\}$, $e :: \{\{[a_1 : \tau_1, \ldots, a_n : \tau_n]\}\}$, $e :: \langle[a_1 : \tau_1, \ldots, a_n : \tau_n]\rangle$, or $e :: [a_1 : \tau_1, \ldots, a_n : \tau_n]$. Giving a set of attributes A, we are sometimes interested in the attributes provided

by an expression e which are not in A. For this complement we use the notation $\overline{A}(e)$ defined as $\mathcal{A}(e) \setminus A$. When e is clear from the context, we use $\overline{A}$ as a shorthand.

Often, we are not only interested in the set of attributes an expression provides, but also in the set of free variables occurring in an expression e. For this, we introduce $\mathcal{F}(e)$ denoting the set of all free variables of e.

Since the subscripts of our algebraic operators can contain arbitrary expressions, they may contain variables or even free variables. Then, there is a need to get bindings for these variables before the subscript expression can be evaluated. These bindings are taken from the argument(s) of the operator. In order to do so, we need a specified binding mechanism. The λ-notation is such a mechanism and can be used e.g., in case of ambiguities. For our purpose, it suffices if we stick to the following convention.

- For an expression e with free variables $\mathcal{F}(e) = \{a_1, \ldots, a_n\}$ and a tuple t with $\mathcal{F}(e) \subseteq \mathcal{A}(t)$ we define $e(t) := e[a_1 \leftarrow t.a_1, \ldots, a_n \leftarrow t.a_n]$.[1] Similarily, we define $e(t_1, \ldots, t_n)$ for more than a single tuple. Note that the attribute names of the t_i have to be distinct to avoid name conflicts.

- For an expression e with only one free variable x, we define $e(t) = e[x \leftarrow t]$.

The mechanism is very much like the standard binding for the relational algebra. Consider for example a select operation $\sigma_{a=3}(R)$, then we assume that a, the free variable of the subscript expression $a = 3$ is bound to the value of the attribute a of the tuples of the relation R. To express this binding explicitly, we would write for a tuple $t \in R$ $(a = 3)(t)$. Since a is an attribute of R and hence of t, by our convention a is replaced by $t.a$, the value of attribute a of tuple t. Since we want to avoid name conflicts right away, we assume that all variable/attribute names used in a query are distinct. This can be achieved in a renaming step.

Application of a function f to arguments e_i is denoted by either regular (e.g., $f(e_1, \ldots, e_n)$) or dot (e.g., $e_1.f(e_2, \ldots, e_n)$) notation. The dot notation is used for type associated methods occurring in the object-oriented context.

Last, we introduce the heavily overloaded symbol $\circ$. It denotes function concatenation and (as a special case) tuple concatenation as well as the concatenation of tuple types to yield a tuple type containing the union of the attributes of the two argument tuple types.

Sometimes it is useful to be able to produce a set/bag/sequence containing only a single tuple with no attributes. This is done by the *singleton scan* operator denoted by $\Box$.

Very often, we are given some database item which is a bulk of other items. Binding these to variables or equivalently, embedding the items into a tuple, we use the notation $e[x]$ for an expression e and a variable/attribute name x. For set valued expressions e, $e[x]$ is defined as $e[x] = \{[x : y] | y \in e\}$. For bags the definition is analogous. For sequence valued expressions e, we define $e[a] = \epsilon$ if e is empty and $e[a] = \langle [a : \alpha(e)] \rangle \oplus \tau(e)[a]$ otherwise.

By id we denote the identity function.

[1] $e[v_1 \leftarrow e_1, \ldots, v_n \leftarrow e_n]$ denotes a substitution of the variables v_i by the expressions e_i within an expression e.

6.2.3 Operator Signatures

We are now ready to define the signatures of the operators of our algebra. Their semantics is defined in a subsequent step. Remember that we consider all operators as being

polymorphic. Hence, their signatures are polymorphic and contain type variables.

$$
\begin{array}{rcl}
\cup & : & \{\tau\}, \{\tau\} \rightarrow \{\tau\} \\
\cap & : & \{\tau\}, \{\tau\} \rightarrow \{\tau\} \\
\setminus & : & \{\tau\}, \{\tau\} \rightarrow \{\tau\} \\
\sigma_p & : & \{\tau\} \rightarrow \{\tau\} \\
 & & \text{if } p : \tau \rightarrow \mathcal{B} \\
\bowtie_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2\} \\
 & & \text{if } \tau_i \leq [], p : \tau_1, \tau_2 \rightarrow \mathcal{B}, \text{ and} \\
 & & \mathcal{A}(\tau_1) \cap \mathcal{A}(\tau_2) = \emptyset \\
\rtimes_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1\} \\
 & & \text{if } \tau_i \leq [], p : \tau_1, \tau_2 \rightarrow \mathcal{B} \\
\triangleright_p & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1\} \\
 & & \text{if } \tau_i \leq [], p : \tau_1, \tau_2 \rightarrow \mathcal{B} \\
\exists\!\!\bowtie_p^{g=c} & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2^+\} \\
 & & \text{if } \tau_1 < [], c :: \tau, \tau_2 < [g : \tau], \\
 & & + \text{ just adds a null value to the domain of } \tau_2, \\
 & & \text{if it does not already contain one,} \\
 & & p : \tau_1, \tau_2 \rightarrow \mathcal{B} \\
< \cdot > & : & \{\tau_1\} || \{\tau_2\} \rightarrow \{\tau_1 \circ \tau_2\} \\
 & & \text{if } \tau_i \leq [], \\
 & & \mathcal{A}(\tau_1) \cap \mathcal{A}(\tau_2) = \emptyset \\
\chi_f & : & \{\tau_1\} \rightarrow \{\tau_2\} \\
 & & \text{if } f : \tau_1 \rightarrow \tau_2 \\
\Gamma_{g;\theta A;f} & : & \{\tau\} \rightarrow \{\tau \circ [g : \tau']\} \\
 & & \text{if } \tau \leq [], f : \{\tau\} \rightarrow \tau', A \subseteq \mathcal{A}(\tau) \\
\Gamma_{g;A_1\theta A_2;f} & : & \{\tau_1\}, \{\tau_2\} \rightarrow \{\tau_1 \circ [g : \tau']\} \\
 & & \text{if } \tau_1 \leq [], f : \{\tau_2\} \rightarrow \tau', A_i \subseteq \mathcal{A}(\tau_i) \text{ for } i = 1, 2 \\
\mu_g & : & \{\tau\} \rightarrow \{\tau'\} \\
 & & \text{if } \tau = [a_1 : \tau_1, \ldots, a_n : \tau_n, g : \{\tau_0\}], \\
 & & \tau_0 \leq [], \\
 & & \tau' = [a_1 : \tau_1, \ldots, a_n : \tau_n] \circ \tau_0, \\
\mu_{g;c} & : & \{\tau\} \rightarrow \{\tau'\} \\
 & & \text{if } \tau = [a_1 : \tau_1, \ldots, a_n : \tau_n, g : \{\tau_0\}], \\
 & & \tau_0 \not\leq [], \\
 & & \tau' = [a_1 : \tau_1, \ldots, a_n : \tau_n] \circ [c : \tau_0], \\
flatten & : & \{\{\tau\}\} \rightarrow \{\tau\} \\
max_{g;m;a\theta;f} & : & \{\tau\} \rightarrow [m : \tau_a, g : \tau_f] \\
 & & \text{if } \tau \leq [a : \tau_a], f : \{\tau\} \rightarrow \tau_f
\end{array}
$$

6.2.4 Selection

Note that in the following definition there is no restriction on the selection predicate. It may contain path expressions, method calls, nested algebraic operators, etc.

$$\begin{aligned}\sigma_p(e) &= \{x|x \in e, p(x)\}\\ \bar{\sigma}_p(e) &= \{\{x|x \in e, p(x)\}\}\end{aligned}$$

6.2.5 Projection

6.2.6 Map

These operators are fundamental to the algebra. As the operators mainly work on sets of tuples, sets of non-tuples (mostly sets of objects) must be transformed into sets of tuples. This is one purpose of the map operator. Other purposes are dereferenciation, method and function application. Our translation process also pushes all nesting into map operators.

The first definition corresponds to the standard map as defined in, e.g., [475]. The second definition concerns the special case of a materialize operator [474, 85]. The third definition handles the frequent case of constructing a set of tuples with a single attribute out of a given set of (non-tuple) values.

$$\begin{aligned}\chi_{e_2}(e_1) &= \{e_2(x)|x \in e_1\}\\ \chi_{a:e_2}(e_1) &= \{y \circ [a : e_2(y)]|y \in e_1\}\\ e[a] &= \{[a : x]|x \in e\}\end{aligned}$$

Note that the oo map operator obviates the need of a relational projection. Sometimes the map operator is equivalent to a simple projection (or renaming). In these cases, we will use π (or ρ) instead of χ.

6.2.7 Join Operators

The algebra features five join operators. Besides the complex join predicate, the first four of them are rather standard: join, semi-join, anti-join and left-outer join are defined similarly to their relational counterparts. One difference is that the left-outer join accepts a default value to be given, instead of null, to one attribute of its right argument.

$$\begin{aligned}e_1 \bowtie_p e_2 &= \{y \circ x|y \in e_1, x \in e_2, p(y,x)\}\\ e_1 \rtimes_p e_2 &= \{y|y \in e_1, \exists x \in e_2, p(y,x)\}\\ e_1 \triangleright_p e_2 &= \{y|y \in e_1, \neg\exists x \in e_2\ p(y,x)\}\\ e_1 \overset{g=c}{\underset{p}{\exists\!\bowtie}} e_2 &= \{y \circ x|y \in e_1, x \in e_2, p(y,x)\} \cup\\ &\quad \{y \circ z|y \in e_1, \neg\exists x \in e_2\ p(y,x), \mathcal{A}(z) = \mathcal{A}(e_2), g \in \mathcal{A}(e_2),\\ &\quad z.g = c, \forall a \in \mathcal{A}(e_2)\ \ (z.a \neq NULL \Longrightarrow a = g)\}\end{aligned}$$

Remember that the function $\mathcal{A}$ used in the last definition returns the set of attributes of a relation.

The last join operator is called *d-join* ($< \cdot >$). It is a join between two sets, where the evaluation of the second set may dependent on the first set. It is used to translate **from** clauses into the algebra. Here, the range definition of a variable may depend on the value of a formerly defined variable. Whenever possible, d-joins are rewritten into standard joins.

$$e_1 < e_2 > \quad = \quad \{y \circ x | y \in e_1, x \in e_2(y)\}$$

Grouping Operators Two grouping operators will be used for unnesting purposes. The first one — called *unary grouping* — is defined on a set and its subscript indicates (i) the attribute receiving the grouped elements (ii) the grouping criterion, and (iii) a function that will be applied to each group.

$$\begin{aligned} \Gamma_{g;\theta A;f}(e) \quad = \quad & \{y.A \circ [g : G] | y \in e, \\ & G = f(\{x | x \in e, x.A \theta y.A\})\} \end{aligned}$$

Note that the traditional nest operator [732] is a special case of unary grouping. It is equivalent to $\Gamma_{g;A=;id}$. Note also that the grouping criterion may be defined on several attributes. Then, A and θ represent sequences of attributes and comparators.

The second grouping operator — called *binary grouping* — is defined on two sets. The elements of the second set are grouped according to a criterion that depends on the elements of the first argument.

$$e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 \quad = \quad \{y \circ [g : G] | y \in e_1, G = f(\{x | x \in e_2, y.A_1 \theta x.A_2\})\}$$

In the sequel, the following abbreviations will be used: $\Gamma_{g;A;f}$ for $\Gamma_{g;A=;f}$, $\Gamma_{g;A}$ for $\Gamma_{g;A;id}$.

New implementation techniques have to be developed for these grouping operators. Obviously, those used for the nest operator can be used for simple grouping when θ stands for equality. For the other cases, implementations based on sorting seem promising. We also consider adapting algorithms for non-equi joins, e.g. those developed for the band-width join [232]. A very promissing approach is the use of θ-tables developed for efficient aggregate processing [188].

Unnest Operators The unnest operators come in two different flavor. The first one is responsible for unnesting a set of tuples on an attribute being a set of tuples itself. The second one unnests sets of tuples on an attribute not being a set of tuples but a set of something else, e.g., integers.

$$\begin{aligned} \mu_g(e) \quad &= \quad \{y.[\mathcal{A}(y) \setminus \{g\}] \circ x | y \in e, x \in y.g\} \\ \mu_{g;c}(e) \quad &= \quad \{y.[\mathcal{A}(y) \setminus \{g\}] \circ [c : x] | y \in e, x \in y.g\} \end{aligned}$$

Flatten Operator The flatten operator flattens a set of sets by unioning the elements of the sets contained in the outer set.

$$flatten(e) \quad = \quad \{y | x \in e, y \in x\}$$

Max Operator The Max operator has a very specific use that will be explained in the sequel. Note that an analogous Min operator can be defined.

$$Max_{g;m;a\theta;f}(e) = [m : max(\{x.a|x \in e\}), g : f(\{x|x \in e, x.a\theta m\})]$$

This definition is a generalization of the *Max* operator as defined in [186]. Since the equivalences don't care whether we use *Max* or *Min*, we write *Agg* to denote either of them.

Remarks Note that, apart from the χ and $flatten$ operations, all these operations are defined on sets of tuples. This guarantees some nice properties among which is the associativity of the join operations. Note also that the operators may take complex expressions in their subscript, therefore allowing nested algebraic expressions. This is the most fundamental feature of the algebra when it comes to express nested queries at the algebraic level. Unnesting is then expressed by algebraic equivalences moving algebraic expression out of the superscript.

The Γ, $flatten$ and Max operations are mainly needed for optimization purposes, as we will see in the sequel, but do not add power to the algebra. Note that a Min operation similar to the Max operation can easily be defined.

The algebra is defined on sets whereas most OBMS also manipulate lists and bags. We believe that our approach can easily be extended by considering lists as set of tuples with an added positional attribute and bags as sets of tuples with an added key attribute.

6.2.8 Linearity and Reorderability

Linearity of Algebraic Operators

As already mentioned, the core argument for optimizability of algebraic expressions is the reorderability of their operators. One could discuss the reorderability of each two operators resulting in n^2 investigations if the number of operators in the algebra is n. In order to avoid this, we introduce a general argument covering most of the cases. This argument is that linearity of operators implies their reorderability easily. Hence, let us first look at the linearity property.

A unary mapping $f : \{\tau\} \to \{\tau'\}$ is called *linear* (or *homomorph*), if and only if

$$\begin{aligned} f(\emptyset) &= \emptyset \\ f(A \cup B) &= f(A) \cup f(B) \end{aligned}$$

If τ, τ' and τ_i are collection types, an n-ary mapping

$$f : \tau_1, \ldots, \tau_{i-1}, \{\tau\}, \tau_{i+1}, \ldots, \tau_n \to \{\tau'\}$$

is called *linear in its i-th argument*, iff for all $e_1, \ldots, e_n, e_i'$

$$\begin{aligned} f(e_1, \ldots, e_{i-1}, \emptyset, e_{i+1}, \ldots, e_n) &= \emptyset \\ f(e_1, \ldots, e_{i-1}, e_i \cup e_i', e_{i+1}, \ldots, e_n) &= f(e_1, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots, e_n) \\ &\quad \cup f(e_1, \ldots, e_{i-1}, e_i', e_{i+1}, \ldots, e_n) \end{aligned}$$

It is called *linear*, if it is linear in all its arguments. Note that if an equivalence with linear mappings on both sides has to be proven, it suffices to proof it for disjunct singletons, i.e. sets with one element only.

The following summarizes the findings on the linearity of the algebraic operators:

$\cup$ ....

$\cap$ ....

$\setminus$ ....

π ....

π^d ....

χ_f is linear.

$$\begin{aligned}\chi_f(\emptyset) &= \emptyset \\ \chi_f(e_1 \cup e_2) &= \{f(x)|x \in e_1 \cup e_2\} \\ &= \{f(x)|x \in e_1\} \cup \{f(x)|x \in e_2\} \\ &= \chi_f(e_1) \cup \chi_f(e_2)\end{aligned}$$

σ is linear (for a proof see [866])

$\bowtie_p$ is linear (for a proof see [866]). Similarly, $\triangleright<$ is linear. $\triangleright$ is linear in its first argument but obviously not in its second.

$\overline{\bowtie}$ is linear in its first argument.

$$\begin{aligned}\emptyset \,\overline{\bowtie}\, {}^{g=c}_{p} e &= \emptyset \\ (e_1 \cup e_2) \,\overline{\bowtie}\, {}^{g=c}_{p} e &= \{y \circ x | y \in e_1 \cup e_2, x \in e, p(y,x)\} \cup \\ &\quad \{y \circ z | y \in e_1 \cup e_2, \neg\exists x \in e\, p(y,x), \mathcal{A}(z) = \mathcal{A}(e), \\ &\quad \forall a\, a \neq g \succ z.a = NULL, a.g = c\} \\ &= (e_1 \,\overline{\bowtie}\, {}^{g=c}_{p} e) \cup (e_2 \,\overline{\bowtie}\, {}^{g=c}_{p} e)\end{aligned}$$

To see that $\overline{\bowtie}$ is not linear in its second argument consider

$$e_1 \,\overline{\bowtie}\, {}^{g=c}_{p} \emptyset = \emptyset \text{ iff } e_1 = \emptyset$$

$< >$ is linear in its first argument:

$$\begin{aligned}\emptyset < e > &= \emptyset \\ (e_1 \cup e_2) < e > &= \{y \circ x | y \in e_1 \cup e_2, x \in e(y)\} \\ &= \{y \circ x | y \in e_1, x \in e(y)\} \cup \{y \circ x | y \in e_2, x \in e(y)\} \\ &= (e_1 < e >) \cup (e_2 < e >)\end{aligned}$$

Note that the notion of linearity cannot be applied to the second (inner) argument of the d-join, since, in general, it cannot be evaluated independently of the first argument. Below, we summarize some basic observations on the d-join.

$\Gamma_{g;A;f}$ is not linear.

Consider the following counterexample:

$$\begin{aligned}\Gamma_{g;a}(\{[a:1,b:1],[a:1,b:2]\}) & \\ &= \{[a:1,g:\{[a:1,b:1],[a:1,b:2]\}]\} \\ &\neq \{[a:1,g:\{[a:1,b:1]\}]\} \cup \{[a:1,g:\{[a:1,b:2]\}]\} \\ &= \Gamma_{g;a}(\{[a:1,b:1]\}) \cup \Gamma_{g;a}(\{[a:1,b:2]\})\end{aligned}$$

μ_g is linear.

$$\begin{aligned}\mu_g(\emptyset) &= \emptyset \\ \mu_g(e_1 \cup e_2) &= \{x.[\overline{g}] \circ y | x \in e_1 \cup e_2, y \in x.g\} \\ &= \{x.[\overline{g}] \circ y | x \in e_1, y \in x.g\} \cup \{x.[\overline{g}] \circ y | x \in e_2, y \in x.g\} \\ &= \mu_g(e_1) \cup \mu_g(e_2)\end{aligned}$$

$\mu_{g;c}$ is also linear. This is shown analogously to the linearity of μ_g.

$flatten$ is linear.

$$\begin{aligned}flatten(e_1 \cup e_2) & \\ &= \{x | y \in e_1 \cup e_2, x \in y\} \\ &= \{x | y \in e_1, x \in y\} \cup \{x | y \in e_2, x \in y\} \\ &= flatten(e_1) \cup flatten(e_2)\end{aligned}$$

Note that the notion of linearity does not apply to the *max* operator, since it does not return a set.

The concatenation of linear mappings is again a linear mapping. Assume f and g to be linear mappings. Then

$$\begin{aligned}f(g(\emptyset)) &= \emptyset \\ f(g(x \cup y)) &= f(g(x) \cup g(y)) \\ &= f(g(x)) \cup f(g(y))\end{aligned}$$

Reorderability Laws

From the linearity considerations of the previous subsection, it is easy to derive reorderability laws.

Let $f : \{\tau_1^f\} \rightarrow \{\tau_2^f\}$ and $g : \{\tau_1^g\} \rightarrow \{\tau_2^g\}$ be two linear mappings. If $f(g(\{x\})) = g(f(\{x\}))$ for all singletons $\{x\}$ then

$$f(g(e)) = g(f(e)) \tag{6.1}$$

For the linear algebraic operators working on sets of tuples, we can replace the semantic condition $f(g(\{x\})) = g(f(\{x\}))$ by a set of syntactic criterions. The

main issue here is to formalize that two operations do not interfere in their consumer/producer/modifier relationship on attributes. In the relational algebra we have the same problem. Nevertheless, it is often neglected there. Consider for example the algebraic equivalence

$$\sigma_p(R \bowtie S) \quad = \quad (\sigma_p(R)) \bowtie S$$

Then, this algebraic equivalence is true only if the predicate p accesses only those attributes already present in R. Now, for our operators we can be sure that $f(g(e)) = g(f(e))$ for singleton sets e, if g does not consume/produce/modify an attribute that f is going to access and if f is not going to consume/produce/modify an attribute that is accessed by g. In fact, most of the conditions attached to the algebraic equivalences given later on concern this point.

We now consider binary mappings. Let f_1 be a binary mapping being linear in its first argument, f_2 a binary mapping being linear in its second argument, and g a unary linear mapping. If $f_1(g(\{x\}), e') = g(f_1(\{x\}, e'))$ for all x and e' then

$$f_1(g(e), e') \quad = \quad g(f_1(e, e')) \tag{6.2}$$

for all e. Again, we can recast the condition $f_1(g(\{x\}), e') = g(f_1(\{x\}), e')$ into a syntactical criterion concerning the consumer/producer/modifier relationship of attributes.

Analogously, if $f_2(e, g(\{x\})) = g(f_2(e, \{x\}))$ for all x and e then

$$f_1(e, g(e')) \quad = \quad g(f_1(e, e')) \tag{6.3}$$

for all e'.

Since the outerjoin is not linear in its second argument, we state at least some reorderability results concerning the reordering of joins with outerjoins since much performance can be gained by chosing a (near-) optimal evaluation order. The results are not original but instead taken from [704] and repeated here for convenience:

$$(e_1 \bowtie_{p_{1,2}} e_2) \mathbin{⟗}_{p_{2,3}} e_3 \quad = \quad e_1 \bowtie_{p_{1,2}} (e_2 \mathbin{⟗}_{p_{2,3}} e_3) \tag{6.4}$$

$$(e_1 \mathbin{⟗}_{p_{1,2}} e_2) \mathbin{⟗}_{p_{2,3}} e_3 \quad = \quad e_1 \mathbin{⟗}_{p_{1,2}} (e_2 \mathbin{⟗}_{p_{2,3}} e_3) \tag{6.5}$$

if $p_{2,3}$ is strong w.r.t. e_2

$$(e_1 \mathbin{⟗}_{p_{1,2}} e_2) \mathbin{⟕}_{p_{2,3}} e_3 \quad = \quad e_1 \mathbin{⟗}_{p_{1,2}} (e_2 \mathbin{⟕}_{p_{2,3}} e_3) \tag{6.6}$$

where an outer join predicate p is strong w.r.t. some expression e_2, if it yields false if all attributes of the relation to be preserved are NULL.

6.2.9 Reordering of joins and outer-joins

Whereas the join operator is commutative and associative, this is no longer true for outer joins. Especially, joins and outer joins together do not behave associatively. Further, not always are outerjoins and joins reorderable. In this subsection we discuss the reorderability of outerjoins. For a full account on the topic see [704, 283, 292]

The occurance of an outer join can have several reasons. First, outer joins are part of the SQL 2 specification. Second, outer joins can be introduced during query rewrite.

L

A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

R

C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

L ⟗ R

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	–	–

L ⟕ R

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
–	–	c_3	d_2	e_2

L ⟖ R

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	–	–
–	–	c_3	d_2	e_2

Figure 6.3: Outer join examples

For example, unnesting nested queries or hierarchical views may result in outer joins. Sometimes, it is also possible to rewrite universal quantifiers to outerjoins [848, 214].

The different outer joins can be defined using the *outer union* operator introduced by Codd [195]. Let R_1 and R_2 be two relations and S_1 and S_2 their corresponding attributes. The outerjoin is then defined by padding the union of the relations with *null* values to the schema $S_1 \cup S_2$:

$$R_1 \overset{+}{\cup} R_2 = (R_1 \times \{null_{S_2 \setminus S_1}\}) \cup (R_2 \times \{null_{S_1 \setminus S_2}\}) \tag{6.7}$$

Given this definition of the *outer union* operator, we can define the outer join operations as follows:

$$R_1 ⟗_p R_2 = R_1 \bowtie_p R_2 \overset{+}{\cup} (R_1 \setminus \pi_{S_1}(R_1 \bowtie_p R_2)) \tag{6.8}$$

$$R_1 ⟕_p R_2 = R_1 \bowtie_p R_2 \overset{+}{\cup} (R_1 \setminus \pi_{S_1}(R_1 \bowtie_p R_2)) \overset{+}{\cup} (R_2 \setminus \pi_{S_2}(R_1 \bowtie_p R_2)) \tag{6.9}$$

$$R_1 ⟖_p R_2 = R_2 ⟕ R_1 \tag{6.10}$$

Each outer join preserves the tuples on the according side. For example, the left-outer join preserves the tuples of the relation on the left-hand side. Figure 6.3 gives example applications of the different outer join operations. A *null* value is indicated by a "-".

Obviously, the left-outer join and the right-outer join are not commutative. To illustrate associativity problems consider the following three relations

R
A
a

S	
B	C
b	-

T
D
d

The results of different left-outer join applications are

R $⟗_{A=B}$ S		
A	B	C
a	–	–

S $⟗_{C=D \vee C=null}$ T		
B	C	D
b	–	c

(R $⟗_{A=B}$ S) $⟗_{C=D \vee C=null}$ T			
A	B	C	D
a	–	–	c

R $⟗_{A=B}$ (S $⟗_{C=D \vee C=null}$ T)			
A	B	C	D
a	–	–	–

Hence, in general $(R ⟗_{p_{RS}} S) ⟗_{p_{ST}} T \neq r ⟗_{p_{RS}} (S ⟗_{p_{ST}} T)$. The problem is that the predicate p_{ST} does not reject *null* values, where a predicate *rejects null values* (or *rejects nulls* for short) in attribute set A, if it evaluates to *false* or *undefined* on every tuple in which all attributes in A are *null*. Using this definition, we have the following identities

$$(R_1 ⋈_{p_{12}} R_2) ⟗_{p_{23}} R_3 = R_1 ⋈_{p_{12}} (R_2 ⟗_{p_{23}} R_3) \quad (6.11)$$

$$(R_1 ⟗_{p_{12}} R_2) ⟗_{p_{23}} R_3 = R_1 ⟗_{p_{12}} (R_2 ⟗_{p_{23}} R_3) \text{ if } p_{23} \text{ rejects nulls on } \mathcal{A}(R_2) \quad (6.12)$$

$$(R_1 ⟕_{p_{12}} R_2) ⟗_{p_{23}} R_3 = R_1 ⟕_{p_{12}} (R_2 ⟗_{p_{23}} R_3) \quad (6.13)$$

$$(R_1 ⟗⟕_{p_{12}} R_2) ⟗⟕_{p_{23}} R_3 = R_1 ⟗⟕_{p_{12}} (R_2 ⟗⟕_{p_{23}} R_3) \text{ if } p_{12} \text{ and } p_{23} \text{ reject nulls on } \mathcal{A}(R_2) \quad (6.14)$$

$$(R_1 ⟗⟕_{p_{12}} R_2) ⟗_{p_{23}} R_3 = R_1 ⟗⟕_{p_{12}} (R_2 ⟗_{p_{23}} R_3) \text{ if } p_{23} \text{ rejects nulls on } \mathcal{A}(R_2) \quad (6.15)$$

Further, we can rewrite an outer join to a regular join whenever null-padded tuples are eliminated by some predicate. Equivalences that allow do so and some further ones are given next.

$$R_1 ⟗_{p_1 \wedge p_2} R_2 = R_1 ⟗_{p_1} (\sigma_{p_2}(R_2)) \text{ if } \mathcal{F}(p_2) \subseteq \mathcal{A}(R_2) \quad (6.16)$$

$$\sigma_{p_1}(R_1 ⟗_{p_2} R_2) = \sigma_{p_1}(R_1) ⟗_{p_2} R_2 \text{ if } \mathcal{F}(p_1) \subseteq \mathcal{A}(R_1) \quad (6.17)$$

$$\sigma_{p_1}(R_1 ⟗_{p_2} R_2) = \sigma_{p_1}(R_1 ⋈_{p_2} R_2) \text{ if } p_1 \text{ rejects nulls on } \mathcal{A}(R_2) \quad (6.18)$$

$$\sigma_{p_1}(R_1 ⟗⟕_{p_2} R_2) = \sigma_{p_1}(R_1 ⟕_{p_2} R_2) \text{ if } p_1 \text{ rejects nulls on } \mathcal{A}(R_2) \quad (6.19)$$

The expression $R_1 ⟗_{p_{12}} (R_2 ⋈_{p_{23}} R_3)$ cannot be reordered given the equivalences so far. It is equal to neither $(R_1 ⟗_{p_{12}} R_2) ⋈_{p_{23}} R_3$ not $(R_1 ⟗_{p_{12}} R_2) ⋈_{p_{23}} R_3$. In order to allow reorderability on this expression, the *generalized outer join* was introduced by Rosenthal and Galindo-Legaria [704]. It preserves attributes for a set $A \subseteq \mathcal{A}(R_1)$ and is defined as

$$R_1 ⟗⟕_p^A R_2 = (R_1 ⋈_p R_2) \overset{+}{\cup} (\pi_A(R_1) \setminus \pi_A(R_1 ⋈_p R_2)) \quad (6.20)$$

With this definition, we have the following equivalences:

$$R_1 ⟗_{p_{12}} (R_2 ⋈_{p_{23}} R_3) = (R_1 ⟗_{p_{12}} R_2) ⟗⟕_{p_{23}}^{\mathcal{A}(R_1)} R_3 \text{ if } p_{23} \text{ rejects nulls on } \mathcal{A}(R_2) \quad (6.21)$$

$$R_1 ⟗⟕_{p_{12}} (R_2 ⋈_{p_{23}} R_3) = (R_1 ⟗⟕_{p_{12}} R_2) ⟗⟕_{p_{23}}^{\mathcal{A}(R_1)} R_3 \text{ if } p_{23} \text{ rejects nulls on } \mathcal{A}(R_2) \quad (6.22)$$

(6.23)

The generalized outer join can be generalized to preserve disjoint sets of attributes in order to derive more equivalences [292].

We only gave the basic equivalences for reordering algebraic expressions containing outer joins. General frameworks for dealing with these expressions in toto are presented in [80, 292].

6.2.10 Basic Equivalences for d-Join and Grouping

The d-join and the grouping operators are not linear. Thus, so far, we do not have any reorderability results of these operators. Since they are further quite new, we give some algebraic equivalences which hold despite the fact that they are not linear. This shows that there exist still some kind of optimization which can be performed in the presence of these operators.

Already at the beginning of this subsection, we mentioned that d-join and unnest are closely related. To be more precise, we state the following:

$$e_1 < e_2 > = \mu_g(\chi_{g:e_2}(e_1)) \tag{6.24}$$

$$\pi_{\mathcal{A}(e_2)} e_1 < e_2 > = \mu_g(\chi_{[g:e_2]}(e_1)) \tag{6.25}$$

Between *flatten* and the d-join there also exists a correspondence:

$$flatten(\chi_{e_2}(e_1)) = \pi_{\mathcal{A}(e_2)}(e_1 < e_2 >) \tag{6.26}$$

The following summarizes basic equivalences on the d-join:

$$e < e > = e \tag{6.27}$$

$$e_1 < e_2 > = e_1 \times e_2 \quad \text{if } \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{6.28}$$

$$e_1 < e_2 > = e_1 \bowtie (e_1 < e_2 >) \tag{6.29}$$

$$e_1 < e_2 >< e_3 > = e_1 < e_3 >< e_2 > \quad \text{if } (\mathcal{A}(e_2) \setminus \mathcal{F}(e_2)) \cap \mathcal{F}(e_3) = \emptyset \text{ and } (\mathcal{A}(e_3) \setminus \mathcal{F}(e_3)) \cap \mathcal{F}(e_2) \neq \emptyset \tag{6.30}$$

$$\pi_{\mathcal{A}(e_1)}(e_1 < e_2 >) = \sigma_{e_2 != \emptyset}(e_1) \tag{6.31}$$

$$\tag{6.32}$$

We call a function

$$f \text{ extending} :\prec\succ \forall x, y : f(x) \circ y \downarrow \Longrightarrow f(x \circ y) = f(x) \circ y$$

We call a function

$$f \text{ restricting} :\prec\succ \forall x, y : f(x) \downarrow, x \circ y \downarrow \Longrightarrow f(x \circ y) = f(x)$$

$\pi_{A':A}$ denotes projection on the attributes A and renaming to A'. Unnesting of opera-

tions burried in the d-join can be performed by applying the following equivalence:

$$
\begin{aligned}
e_1 < \sigma_p(e_2) > &= \sigma_p(e_1 < e_2 >) && (6.33)\\
e_1 < \sigma_{A_1\theta A_2}(e_2) > &= e_1 \bowtie_{A_1\theta A_2} e_2 && (6.34)\\
&\quad \text{if } \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, A_i \subseteq \mathcal{A}(e_i)\\
\pi_{A':A}(e) < f(\sigma_{A=A'}(e)) > &= \mu_g(\pi_{A:A'}(\Gamma_{g;A;f}(e))) && (6.35)\\
&\quad \text{if } \mathcal{A} \subseteq \mathcal{A}(\lceil)\\
e_1 < e_2 \bowtie e_3 > &= (e_1 < e_2 >) \bowtie e_3 && (6.36)\\
&\quad \text{if } \mathcal{F}(e_3) \cap \mathcal{A}(e_1) = \emptyset\\
e_1 < \chi_f(e_2) > &= \chi_f(e_1 < e_2 >) && (6.37)\\
&\quad \text{if } f \text{ extending}\\
\pi_A(e_1 < \chi_f(e_2) >) &= \pi_A(\chi_f(e_1 < e_2 >)) && (6.38)\\
&\quad \text{if } A \subseteq \mathcal{A}(\chi_f(e_2)), \text{ and } f \text{ restricting}\\
e_1 < \mu_g(e_2) > &= \mu_g(e_1 < e_2 >) && (6.39)\\
e_1 < \mu_{g;c}(e_2) > &= \mu_{g;c}(e_1 < e_2 >) && (6.40)
\end{aligned}
$$

For f_1 being σ or χ and the non-linear unary Γ we still have

$$
\begin{aligned}
f_1(\Gamma_{g;=A;f_2}(e)) &= \Gamma_{g;=A;f_2}(f_1(e)) && (6.41)\\
&\quad \text{if } \mathcal{F}(f_1) \cap (A \cup \mathcal{A}(f_2) \cup \{g\}) = \emptyset, (A \cup \mathcal{F}(f_2)) \subseteq \mathcal{A}(f_1(e))
\end{aligned}
$$

This equivalence should, e.g., be used form left to right for selections, in order to reduce the cardinality for the Γ operator. For the binary Γ we have

$$
f_1(e_1\Gamma_{g;A_1\theta A_2;f_2}e_2) = f_1(e_1)\Gamma_{g;A_1\theta A_2;f_2}e_2 \quad (6.42)
$$

since the binary Γ is linear in its first argument.

Lately, work has been reported on the reordering of grouping and join operations despite the fact that grouping is not linear [141, 910, 911, 912, 913]. Since pushing grouping inside join operations can result in a tremendious speed up, let us sketch at least the most basic equivalence:

$$
\Gamma_{g;A;agg(e)}(e_1 \bowtie e_2) \equiv e_1 \bowtie (\Gamma_{g;A;agg(e)}(e_2))
$$

This sketched equivalence only holds under certain conditions. For details on the conditions and the correctness proof see [912] and Section 21.7.1.

6.2.11 Simplifying Expressions Containing Joins

Since the join operation is very expensive, it makes sense to investigate expressions containing joins very intensely in order to discover optimization potential. In this subsubsection, we do so.

Sometimes, redundant joins can be eliminated:

$$\pi_{\mathcal{A}(e_2)}(e_1 \bowtie_{A_1=A_2} e_2) = e_2 \quad \text{if } A_1 = \mathcal{A}(e_1), \pi_{A_2}(e_2) \subseteq \pi_{A_2:A_1}(e_1) \tag{6.43}$$

$$e_1 \overset{g:c}{\underset{A_1=A_{2,3}}{\exists\!\!\bowtie}}(e_2 \bowtie_{A_2=A_3} e_3) = e_1 \overset{g:c}{\underset{A_1=A_3}{\exists\!\!\bowtie}} e_3 \quad \text{if } A_1 \subseteq \mathcal{A}(e_1), A_2 \subseteq \mathcal{A}(e_2), A_3 \subseteq \mathcal{A}(e_3), A_{2,3} \subseteq \mathcal{A}(e_2) \cup \mathcal{A}(e_3), A_2' = \mathcal{A}(e_1) \cup \mathcal{A}(e_3), \pi_{A_2:A_2'}(e_1 \bowtie_{A_1=A_3} e_3) \subseteq e_2 \tag{6.44}$$

$$e_1\Gamma_{g;A_1=A_{2,3};f}(e_2 \bowtie_{A_2=A_3} e_3) = e_1\Gamma_{g;A_1=A_2;f}e_3 \quad \text{if } A_1 \subseteq \mathcal{A}(e_1), A_2 \subseteq \mathcal{A}(e_2), A_3 \subseteq \mathcal{A}(e_3), A_{2,3} \subseteq \mathcal{A}(e_2) \cup \mathcal{A}(e_3), A_2' = \mathcal{A}(e_1) \cup \mathcal{A}(e_3), \pi_{A_2:A_2'}(e_1 \bowtie_{A_1=A_3} e_3) \subseteq e_2 \tag{6.45}$$

Equivalences 6.44 and 6.45 bear similarity to equivalence EJA of [866] (p. 107), where the outer-aggregation is used instead of semi-join and binary grouping, respectively. Note that for checking conditions of the form $\pi_{A_2}(e_2) \subseteq \pi_{A_2:A_1}(e_1)$ subtyping implying subsetrelationships on type extensions plays a major role in object bases.

The following shows how to turn an outerjoin into a join:

$$e_1 \overset{g:c}{\underset{p}{\exists\!\!\bowtie}} e_2 = e_1 \bowtie_p e_2 \quad \text{if } \neg p(c) \tag{6.46}$$

$$\sigma_{p_s}(e_1 \overset{g:c}{\underset{p_j}{\exists\!\!\bowtie}} e_2) = \sigma_{p_s}(e_1 \bowtie_{p_j} e_2) \quad \text{if } \neg p_s(c) \tag{6.47}$$

$$\sigma_{f_1\theta f_2}(e_1 \overset{g:c}{\underset{p_j}{\exists\!\!\bowtie}} e_2) = \sigma_{f_1\theta f_2}(e_1 \bowtie_{p_j} e_2) \quad \text{if } \neg(f_1\theta f_2)(c) \tag{6.48}$$

We can easily check these conditions if some predicate $(f_1\theta f_2)(c)$ is—after inserting c for the free variable—, $i\theta 0$ with i constant, or $f_1 \in \emptyset$, or a similar simple form. An application of these equivalences can sometimes be followed by a removal of the join.

Note, that the latter equivalence depends on the knowledge we have on the selection predicate. Note also, that the special outerjoin is only introduced by unnesting nested χ operations. Hence, we could combine the equivalences introducing the outerjoin and replacing it by a join into one. In case we have even more information on the selection predicate than above, more specifically, if it depends on a *max* or *min* aggregate, we can do so in a very efficient way:

$$\sigma_{a=m}(e_1)[m : agg(\chi_b(e_2))] = Agg_{g;m;a}(e_1).g \quad \text{if } \pi_a(e_1) = \pi_b(e_2) \tag{6.49}$$

$$\chi_{g:\sigma_{a=m}(e_2)}(\chi_{m:agg(e_1)}(e)) = \chi_{Agg_{g;m;a}(e_1)}(e) \quad \text{if } \pi_a(e_1) = \pi_b(e_2) \tag{6.50}$$

6.2.12 Reordering Joins and Grouping

Introduction

In general, join and grouping operations are not reorderable. Consider the following relations R and S

R	A	B
	a	5
	a	6

S	A	C
	a	7
	a	8

Joining these relations R and S results in

$R \bowtie S$	A	B	C
	a	5	7
	a	5	8
	a	6	7
	a	6	8

Applying $\Gamma_{A;count(*)}$ to R and $R \bowtie S$ yields

$\Gamma_{A;count(*)}\,(R)$	A	count $(*)$
	a	2

$\Gamma_{A;count(*)}\,(R \bowtie S)$	A	count $(*)$
	a	4

Compare this to the result of $\Gamma_{A;count(*)}(R) \bowtie S$:

$\Gamma_{A;count(*)}\,(R) \bowtie S$	A	count $(*)$	C
	a	2	7
	a	2	8

Hence $\Gamma_{A;count(*)}\,(R) \bowtie S \neq \Gamma_{A;count(*)}\,(R \bowtie S)$.

Since grouping and join operations are in general not reorderable, it is important that a query language determines the order of grouping and join operators properly. In SQL, the grouping operator is applied after the join operators of a query block.

For example, given the relations schemata

Emp (eid, name, age, salary) and
Sold (sid, eid, date, product_id, price)

and the query

select	e.eid, **sum** (s.price) as amount
from	Emp e, Sold s
where	e.eid = s.eid **and**
	s.date **between** "2000-01-01" **and** "2000-12-31"
group by	s.eid, s.name

results in the algebraic expression

$$\Pi_{e.eid,amount}\left(\Gamma_{s.eid;amount:\mathbf{sum}(s.price)}\left(Emp[e] \bowtie_{e.eid=s.eid} \sigma_p\left(Sold[s]\right)\right)\right)$$

where p denotes

$$s.date \geq \text{`}2000-01-01' \wedge s.date \leq \text{`}2000-12-31'$$

Figure 20.1 (a) shows this plan graphically. Note that the grouping operator is executed last in the plan.

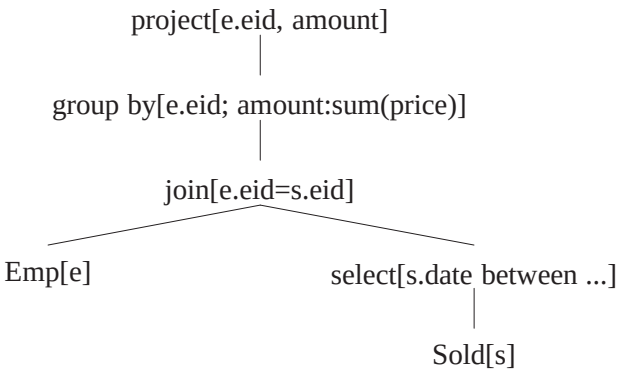
Now consider the plan where we push the grouping operator down:

$$\Pi_{e.eid,amount}\left(Emp[e] \bowtie_{e.eid=s.eid} \left(\Gamma_{s.eid;amount:\mathbf{sum}(s.price)}\left(\sigma_p\left(Sold[s]\right)\right)\right)\right)$$

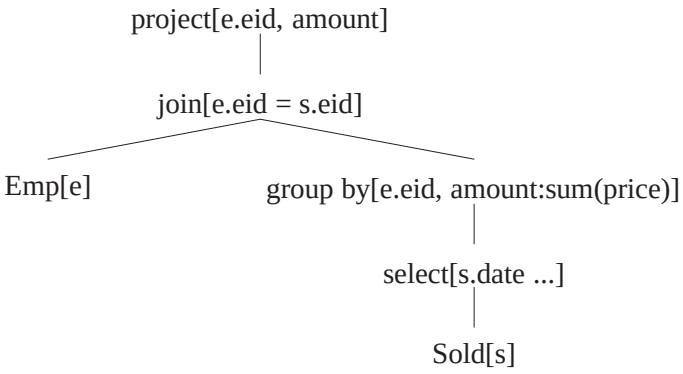
This plan (see also Figure 20.1 (b)) is equivalent to the former plan. Moreover, if the grouping operator strongly reduces the cardinality of

$$\sigma_{s.date \geq ...}(Sold[s])$$

because every employee sells many items, then the latter plan might become cheaper since the join inputs are smaller than in the former plan. This motivates the search for conditions under which join and grouping operators can be reordered. Several papers discuss this reorderability [142, 910, 911, 912, 913]. We will summarize their results in subsequent subsections.



(a)



(b)

Figure 6.4: Two equivalent plans

Lazy and eager group by

Lazy group by pulls a group operator up over a join operator [910, 911, 912, 913]. Eager group by does the opposite.

Consider the query:

select[**all** | **distinct**]	$A, \vec{F}(B)$
from	R, S
where	$p_R \wedge p_S \wedge p_{R,S}$
group by	G

with

$$G = G_R \cup G_S, G_R \subseteq \mathcal{A}(R), G_S \subseteq \mathcal{A}(S),$$

$$\mathcal{F}(p_R) \subseteq \mathcal{A}(R), \mathcal{F}(p_S) \subseteq \mathcal{A}(S)$$

$$\mathcal{F}(p_{R,S}) \subseteq \mathcal{A}(R) \cup \mathcal{A}(S)$$

$$B \subseteq \mathcal{A}(R) \qquad A = A_R \cup A_S, A_R \subseteq G_R, A_S \subseteq G_S$$

$$\alpha_R = G_R \cup \mathcal{F}(p_{R,S}) \quad \setminus \mathcal{A}(S) \quad \kappa_R \text{ key of } R$$

$$\alpha_S = G_S \cup \mathcal{F}(p_{R,S}) \quad \setminus \mathcal{A}(R) \quad \kappa_S \text{ key of } S$$

We are interested in the conditions under which the query can be rewritten into

select[**all** | **distinct**]	A, FB	
from	R', S'	
where	$p_{R,S}$	
with	$R'(\alpha_R, FB) \equiv$	
	select all	$\alpha_R, \vec{F}(B)$ as FB
	from	R
	where	p_R
	group by	α_R
and	$S'(\alpha_S) \equiv$	
	select all	α_R
	from	S
	where	p_S

The following equivalence expresses this rewrite in algebraic terms.

$$\Pi^{[d]}_{A,F}\left(\Gamma_{G;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\bowtie_{p_{R,S}}\sigma_{p_S}(S)\right)\right)\equiv$$

$$\Pi^{[d]}_{A,F}\left(\Gamma_{\alpha_R;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\right)\bowtie_{p_{R,S}}\sigma_{p_S}(S)\right)$$

holds iff in $\sigma_{p_R\wedge p_S\wedge p_{R,S}}(R\times S)$ the following functional dependencies hold:
$FD_1 : G\rightarrow\alpha_R$
$FD_2 : \alpha_R, G_S\rightarrow\kappa_S$
Note that since $G_S\subseteq G$, this implies $G\rightarrow\kappa_S$.

FD_2 implies that for any group there is at most one join partner in S. Hence, each tuple in $\Gamma_{\alpha_R;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\right)$ contributes at most one row to the overall result.

FD_1 ensures that each group of the expression on the left-hand side corresponds to at most one group of the group expression on the right-hand side.

We now consider queries with a **having** clause.

In addition to the assumptions above, we have that the tables in the **from** clause can be partitioned into R and S such that R contains all aggregated columns of both the **select** and the **having** clause. We further assume that conjunctive terms in the **having** clause that do not contain aggregate functions have been moved to the **where** clause.

Let the predicate of the **having** clause have the form $H_R\wedge H_0$ where $H_R\subseteq\mathcal{A}(R)$ and $H_0\subseteq R\cup S$ where H_0 only contains non-aggregated columns from S.

We now consider all queries of the form

select[**all** | **distinct**]	$A,\vec{F}(B)$
from	R,S
where	$p_R\wedge p_S\wedge p_{R,S}$
group by	G
having	$H_0\left(\vec{F_0}(B)\right)\wedge H_R\left(\vec{F_R}(B)\right)$

where $\vec{F_0}$ and $\vec{F_R}$ are vectors of aggregate functions on the aggregated columns B.

An alternative way to express such a query is

select[**all** | **distinct**]	G,FB
from	R',S
where	$c_S\wedge c_{R,S}\wedge H_0(F_0B)$
where	$R'\left(\alpha_R,FB,F_0B\right)\equiv$
select all	$\alpha_R,\vec{F}(B)$ as $FB,\vec{F_0}(B)$ as F_0B
from	R
where	c_R
group by	α_R
having	$H_R\left(\vec{F_R}(B)\right)$

The according equivalence is [912]:

$$\Pi_{G,F}\left(\sigma_{H_R \wedge H_0}\left(\Gamma_{G;F:\vec{F}(B),F_R:\vec{F_R}(B),F_0:\vec{F_0}(B)}\left(\sigma_{p_R \wedge p_S \wedge p_{R,S}}\left(R \times S\right)\right)\right)\right)$$
$$\equiv$$
$$\Pi_{G,F}\left(\sigma_{p_{R,S} \wedge p_S \wedge H_0(F_0)}\right)\left(\Pi_{G,F,F_0}\left(\sigma_{H_R}\left(\Gamma_{G;F:\vec{F}(B),F_R:\vec{F_R}(B)F_0:\vec{F_0}(B)}(R)\right)\right) \times S\right)$$

Coalescing Grouping

In this subsection we introduce *coalescing grouping* which slightly generalizes *simple coalescing grouping* as introduced in [142].

We first illustrate the main idea by means of an example.
Given two relation schemes

Sales (pid, deptid, total_price)
Department (did, name, region)

the query

select	region, **sum** (total_price) as s
from	Sales, Department
where	did = deptid
group by	region

is straightforwardly translated into the following algebraic expression:

$$\Gamma_{region;s:sum(total_price)}(\text{Sales} \bowtie_{deptid=did} \text{Department})$$

Note that Equivalence **??** cannot be applied here. However, if there are many sales performed by a department, it might be worth reducing the cardinality of the left join input by introducing an additional group operator. The result is

$$\Gamma_{region;s=sum(s')}\left(\Gamma_{deptid;s':sum(total_price)}(\text{Sales}) \bowtie_{deptid=did} \text{Department}\right)$$

Note that we must keep the outer grouping.

That is, we introduced an additional group operator to reduce the cardinality of sales. This way, all subsequent joins (only one in this case) become cheaper and the additional group operator may result in a better plan.

We have the following restrictions for this subsection:

1. There are no NULL-values allowed for attributes occurring in the query.

2. All queries are of the form **select all**.
   That is **select distinct** is not allowed.

3. All aggregate functions agg must fulfill $\text{agg}s_1 \cup s_2 = \text{agg}\{\text{agg}(s_1), \text{agg}(s_2)\}$ for bags s_1 and s_2.
   This has two consequences:

   - Allowed are only sum, min, max. Not allowed are avg and count.

- For any allowed aggregate function we only allow for **agg(all ...)**. Forbidden is **agg(distinct ...)**.

4. The query is a single-block conjunctive query with no **having** and no **order by** clause.

The above transformation is an application of the following equivalence, where R_1 and R_2 can be arbitrary algebraic expressions:

$$\Gamma_{G;A}\left(R_1 \bowtie_p R_2\right) \equiv \Gamma_{G;A_2}\left(\Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2\right) \tag{6.51}$$

with

$$\begin{aligned} A &= A_1 : agg_1\left(e_1\right), \ldots, A_n : agg_n\left(e_n\right) \\ A_1 &= A_1^1 : agg_1^1\left(e_1\right), \ldots, A_n^1 : agg_n^1\left(e_n\right) \\ A_2 &= A_1 : agg_1^2\left(A_1^1\right), \ldots, A_n : agg_n^2\left(A_n^2\right) \end{aligned}$$

$$G_1 = \left(\mathcal{F}\left(p\right) \cup G\right) \cap \mathcal{A}\left(R_1\right)$$

Further, the following condition must hold for all $i (1 \leq i \leq n)$:

$$agg_i\left(\bigcup_k S_k\right) = agg_i^2\left(\bigcup_k \{agg_i^1\left(S_i\right)\}\right)$$

In the above example, we had $agg_1 = agg_1^1 = agg_1^2 =$ **sum**.

We now prove the correctness of Equivalence 20.1.

Proof:
First, note that

$$R_1 \bowtie_p R_2 = \bigcup_{t_2 \in R_2} R_1 \bowtie_p \{t_2\} \tag{6.52}$$

Second, note that for a given t_2

$$\begin{aligned} \Gamma_{G;A}\left(R_1[t_1]\right) \bowtie_p \{t_2\} &= \sigma_{p(t_1 \circ t_2)}\left(\Gamma_{G;A}\left(R_1\left[t_1\right]\right)\right) \\ &= \Gamma_{G;A}\left(\sigma_{p(t_1 \circ t_2)}\left(R_1\left[t_1\right]\right)\right) \\ &= \Gamma_{G;A}\left(R_1\left[t_1\right] \bowtie_p \{t_2\}\right) \end{aligned} \tag{6.53}$$

holds where we have been a little sloppy with t_1. Applying (20.2) and (20.3) to $\Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2$, the inner part of the right-hand side of the equivalence yields:

$$\begin{aligned} \Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2 &= \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p \{t_2\} \\ &= \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}\left(R_1 \bowtie_p \{t_2\}\right) \end{aligned} \tag{6.54}$$

Call the last expression X.

Then the right-hand side of our equivalence becomes

$$\begin{aligned}\Gamma_{G;A_2}(X) &= \{t \circ a_2 | t \in \Pi_G(X), a_2 = \left(A_1 : a_1^2, \ldots, A_n : a_n^2\right), \\ &\quad a_i^2 = agg_i^2\left(\{s.A_i^1 | s \in X, S|_G = t\}\right)\}\end{aligned} \tag{6.55}$$

Applying (20.2) to the left-hand side of the equivalence yields:

$$\Gamma_{G;A}\left(R_1 \bowtie_p R_2\right) = \Gamma_{G;A}\left(\bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p\left(t_1 \circ t_2\right)\}\right) \tag{6.56}$$

Abbreviate $\bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p\left(t_1 \circ t_2\right)\}$ by Y.
Applying the definition of $\Gamma_{G;A}$ yields:

$$\begin{aligned}\{t \circ a \mid\ & t \in \Pi_G(Y), a = \left(A_1 : e_1, \ldots, A_n : e_n\right), \\ & a_i = agg_i\left(\{e_i(s) | s \in Y, S|_G = t\}\right)\}\end{aligned} \tag{6.57}$$

Compare (20.5) and (20.7). Since $\Pi_G(X) = \Pi_G(Y)$, they can only differ in their values of A_i.

Hence, it suffices to prove that $a_i^2 = a_i$ for $1 \leq i \leq n$ for any given t.

$$\begin{aligned}
a_i^2 &= agg_i^2(\{s.A_i^1 | s \in X, S|_G = t\}) \\
&= agg_i^2(\{s.A_i^1 | s \in \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}(R_1 \bowtie_p \{t_2\}), S|_G = t\}) \\
&= agg_i^2(\{s.A_i^1 | s \in \bigcup_{t_2 \in R_2} \{t_1 \circ t_2 \circ a_1 | t_1 \in \Pi_{G_1}(R_1), p(t_1 \circ t_2), \\
&\qquad\qquad a_1 = (A_1^1 : a_1^1, \ldots, A_n^1 : a_n^1) \\
&\qquad a_i^1 = agg_i^1(\{e_i(s_1 \circ t_2) | s_1 \in R_1, S_1|_{G_1 = t_1}, p(s_1, t_2)\}) \\
&\qquad S|_G = t\}\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(s_1 \circ t_2) | t_1 \in \Pi_{G_1}(R_1), p(t_1 \circ t_2), s_1 \in R_1, S_1|_{G_1} = t_1, \\
&\qquad\qquad p(s_1, t_1), t_1 \circ t_2|_G = t\})\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(s_1 \circ t_2) | s_1 \in R_1, s_1 \circ t_2|_G = t, p(s_1 \circ t_2)\})\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(t_1 \circ t_2) | t_1 \in R_1, t_1 \circ t_2|_G = t, p(t_1 \circ t_2)\})\}) \\
&= agg_i(\bigcup_{t_2 \in R_2} \{e_i(t_1 \circ t_2) | t_1 \in R_1, p(t_1 \circ t_2), t_1 \circ t_2|_G = t\}) \\
&= agg_i(\{e_i(s) | s \in \bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p(t_1 \circ t_2)\}, S|_G = t\}) \\
&= a_i
\end{aligned}$$

Equivalence 20.1 can be used to add an additional coalescing grouping in front of any of a sequence of joins. Consider the schematic operator tree in Figure 20.2(a). It is equivalent to the one in (b), which in turn is equivalent to the one in (c) if the preconditions of Equivalence 20.1 hold. Performing a similar operation multiple times, any of the join operations can be made to be preceded by a coalescing grouping.

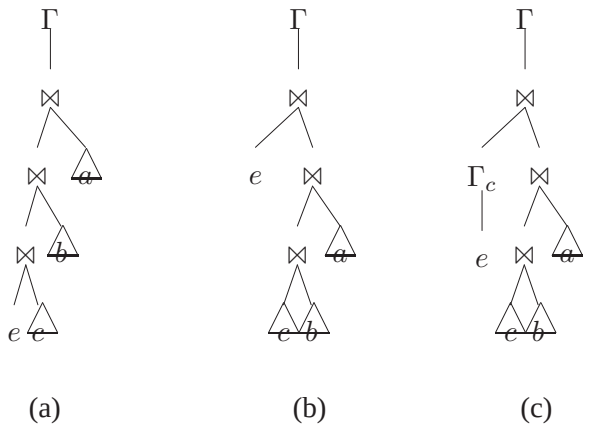


Figure 6.5: Applications of coalescing grouping

6.2.13 ToDo

[688]

6.3 Logical Algebra for Sequences

6.3.1 Introduction

The algebra (NAL) we use here extends the SAL-Algebra [66] developed by Beeri and Tzaban. SAL is the order-preserving counterpart of the algebra used in [186, 187] and in this book.

SAL and NAL work on sequences of sets of variable bindings, i.e., sequences of unordered tuples where every attribute corresponds to a variable. We allow nested tuples, i.e. the value of an attribute may be a sequence of tuples. Single tuples are constructed by using the standard $[\cdot]$ brackets. The concatenation of tuples and functions is denoted by $\circ$. The set of attributes defined for an expression e is defined as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$.

The projection of a tuple on a set of attributes A is denoted by $|_A$. For an expression e_1 possibly containing free variables, and a tuple e_2, we denote by $e_1(e_2)$ the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by e_2. Of course this requires $\mathcal{F}(e_1) \subseteq \mathcal{A}(e_2)$. For a set of attributes we define the tuple constructor $\perp_A$ such that it returns a tuple with attributes in A initialized to NULL.

For sequences e we use $\alpha(e)$ to denote the first element of a sequence. We identify single element sequences and elements. The function τ retrieves the tail of a sequence and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ. As a first application, we construct from a sequence of non-tuple values e a sequence of tuples denoted by $e[a]$. It is empty if e is empty. Otherwise $e[a] = [a : \alpha(e)] \oplus \tau(e)[a]$.

By *id* we denote the identity function. In order to avoid special cases during the translation of XQuery into the algebra, we use the special algebraic operator ($\boxdot$) that returns a singleton sequence consisting of the empty tuple, i.e., a tuple with no attributes.

We will only define order-preserving algebraic operators. For the unordered counterparts see [187]. Typically, when translating a more complex XQuery into our algebra, a mixture of order-preserving and not order-preserving operators will occur. In order to keep the paper readable, we only employ the order-preserving operators and use the same notation for them that has been used in [186, 187] and SAL [66].

Again, our algebra will allow nesting of algebraic expressions. For example, within a selection predicate of a select operator we allow the occurrence of further nested algebraic expressions. Hence, a join within a selection predicate is possible. This simplifies the translation procedure of nested XQuery expressions into the algebra. However, nested algebraic expressions force a nested loop evaluation strategy. Thus, the goal of the paper will be to remove nested algebraic expressions. As a result, we perform unnesting of nested queries not at the source level but at the algebraic level. This approach is more versatile and less error-prone.

6.3.2 Algebraic Operators

We define the algebraic operators recursively on their input sequences. For unary operators, if the input sequence is empty, the output sequence is also empty. For binary operators, the output sequence is empty whenever the left operand represents an empty sequence.

The order-preserving selection operator is defined as

$$\hat{\sigma}_p(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ \alpha(e) \oplus \hat{\sigma}_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \hat{\sigma}_p(\tau(e)) & \text{else} \end{cases}$$

For a list of attribute names A we define the projection operator as

$$\hat{\Pi}_A(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ \alpha(e)|_A \oplus \hat{\Pi}_A(\tau(e)) & \text{else} \end{cases}$$

We also define a duplicate-eliminating projection $\hat{\Pi}^D_A$. Besides the projection, it has similar semantics as the `distinct-values` function of XQuery: it does not preserve order. However, we require it to be deterministic and idempotent. Sometimes we just want to eliminate some attributes. When we want to eliminate the set of attributes A, we denote this by $\hat{\Pi}_{\overline{A}}$. We use $\hat{\Pi}$ also for renaming attributes. Then we write $\hat{\Pi}_{A':A}$. The attributes in A are renamed to those in A'. Attributes other than those in A remain untouched.

The map operator is defined as follows:

$$\hat{\chi}_{a:e_2}(e_1) := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ \alpha(e_1) \circ [a : e_2(\alpha(e_1))] \oplus \hat{\chi}_{a:e_2}(\tau(e_1)) & \text{else} \end{cases}$$

It extends a given input tuple $t_1 \in e_1$ by a new attribute a whose value is computed by evaluating $e_2(t_1)$. For an example see Figure **??**.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$\hat{\chi}_{a:\hat{\sigma}_{A_1=A_2}(R_2)}(R_1) =$

A_1	a
1	$\langle [1,2],[1,3] \rangle$
2	$\langle [2,4],[2,5] \rangle$
3	$\langle\ \rangle$

Figure 6.6: Example for Map Operator

We define the cross product of two tuple sequences as

$$e_1 \hat{\times} e_2 := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ (\alpha(e_1) \hat{\overline{\times}} e_2) \oplus (\tau(e_1) \hat{\times} e_2) & \text{else} \end{cases}$$

where

$$e_1 \hat{\overline{\times}} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \hat{\overline{\times}} \tau(e_2)) & \text{else} \end{cases}$$

We are now prepared to define the join operation on ordered sequences:

$$e_1 \hat{\bowtie}_p e_2 := \sigma_p(e_1 \hat{\times} e_2)$$

We define the semijoin as

$$e_1 \hat{\ltimes}_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \hat{\ltimes}_p e_2) & \text{if } \exists x \in e_2 \;\; p(\alpha(e_1) \circ x) \\ \tau(e_1) \hat{\ltimes}_p e_2 & \text{else} \end{cases}$$

and the anti-join as

$$e_1 \hat{\triangleright}_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \hat{\triangleright}_p e_2) & \text{if } \nexists x \in e_2 \;\; p(\alpha(e_1) \circ x) \\ (\tau(e_1) \hat{\triangleright}_p e_2) & \text{else} \end{cases}$$

The left outer join, which will play an essential role in unnesting, is defined as $e_1 \hat{\overline{\bowtie}}_p^{g:e} e_2 :=$

$$\begin{cases} (\alpha(e_1) \hat{\bowtie}_p e_2) \oplus (\tau(e_1) \hat{\overline{\bowtie}}_p^{g:e} e_2) & \text{if } (\alpha(e_1) \hat{\bowtie}_p e_2) \neq \epsilon \\ (\alpha(e_1) \circ \bot_{\mathcal{A}(e_2) \setminus \{g\}} \circ [g : e]) & \text{else} \\ \quad \oplus (\tau(e_1) \hat{\overline{\bowtie}}_p^{g:e} e_2) & \end{cases}$$

where $g \in \mathcal{A}(e_2)$. Our definition deviates slightly from the standard left outer join operator, as we want to use it in conjunction with grouping and (aggregate) functions. Consider the relations R_1 and R_2 in Figure **??**. If we want to join R_1 (via left outer join) to R_2^{count} that is grouped by A_2 with counted values for B, we need to be able to

handle empty groups (for $A_1 = 3$). e defines the value given to attribute g for values in e_1 that do not find a join partner in e_2 (in this case 0).

We define the dependency join (d-join for short) as

$$e_1\hat{<}e_2\hat{>} := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ \alpha(e_1)\hat{\overline{\times}}e_2(e_1) \oplus \tau(e_1)\hat{<}e_2\hat{>} & \text{else} \end{cases}$$

Let $\theta \in \{=, \leq, \geq, <, >, \neq\}$ be a comparison operator on atomic values. The grouping operator which produces a sequence-valued new attribute containing "the group" is defined by using a binary grouping operator.

$$\hat{\Gamma}_{g;\theta A;f}(e) := \hat{\Pi}_{A:A'}(\hat{\Pi}^D_{A':A}(\hat{\Pi}_A(e))\hat{\Gamma}_{g;A'\theta A;f}e)$$

where the binary grouping operator (sometimes called nest-join [800]) is defined as

$$e_1\hat{\Gamma}_{g;A_1\theta A_2;f}e_2 := \begin{cases} \epsilon & \text{if } e_1 = \epsilon \\ \alpha(e_1) \circ [g : G(\alpha(e_1))] \oplus (\tau(e_1)\hat{\Gamma}_{g;A_1\theta A_2;f}e_2 & \end{cases}$$

Here, $G(x) := f(\sigma_{x|_{A_1}\theta A_2}(e_2))$ and function f assigns a meaningful value to empty groups. See also Figure **??** for an example. The unary grouping operator processes a single relation and obviously groups only on those values that are present. The binary grouping operator works on two relations and uses the left hand one to determine the groups. This will become important for the correctness of the unnesting procedure.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$\hat{\Gamma}_{g;=A_2;count}(R_2) = R_2^{count}$

A_2	g
1	2
2	2

$\hat{\Gamma}_{g;=A_2;id}(R_2) = R_2^g$

A_2	g
1	$\langle[1,2],[1,3]\rangle$
2	$\langle[2,4],[2,5]\rangle$

$R_1\hat{\Gamma}_{g;A_1=A_2;id}(R_2) = R_{1,2}^g$

A_1	g
1	$\langle[1,2],[1,3]\rangle$
2	$\langle[2,4],[2,5]\rangle$
3	$\langle\,\rangle$

Figure 6.7: Examples for Unary and Binary $\hat{\Gamma}$

Given a tuple with a sequence-valued attribute, we can unnest it using the unnest operator defined as

$$\hat{\mu}_g(e) := \begin{cases} \epsilon & \text{if } e = \epsilon \\ (\alpha(e)|_{\overline{\{g\}}}\hat{\times}\alpha(e).g) \oplus \hat{\mu}_g(\tau(e)) & \text{else} \end{cases}$$

where $e.g$ retrieves the sequence of tuples of attribute g. In case that g is empty, it returns the tuple $\perp_{\mathcal{A}(e.g)}$. (In our example in Figure **??**, $\hat{\mu}_g(R_2^g) = R_2$.)

We define the unnest map operator as follows:

$$\hat{\Upsilon}_{a:e_2}(e_1) := \hat{\mu}_g(\chi_{g:e_2[a]}(e_1))$$

This operator is mainly used for evaluating XPath expressions. Since this is a very complex issue [329, 331, 407], we do not delve into optimizing XPath evaluation but instead take an XPath expression occurring in a query as it is and use it in the place of e_2. Optimized translation of XPath is orthogonal to our unnesting approach and not covered in this paper. The interested reader is referred to [407].

6.3.3 Equivalences

To acquaint the reader with ordered sequences, we state some familiar equivalences that still hold.

$$\hat{\sigma}_{p_1}(\hat{\sigma}_{p_2}(e)) = \hat{\sigma}_{p_2}(\hat{\sigma}_{p_1}(e)) \tag{6.58}$$
$$\hat{\sigma}_{p}(e_1 \hat{\times} e_2) = \hat{\sigma}_{p}(e_1) \hat{\times} e_2 \tag{6.59}$$
$$\hat{\sigma}_{p}(e_1 \hat{\times} e_2) = e_1 \hat{\times} \hat{\sigma}_{p}(e_2) \tag{6.60}$$
$$\hat{\sigma}_{p_1}(e_1 \hat{\bowtie}_{p_2} e_2) = \hat{\sigma}_{p_1}(e_1) \hat{\bowtie}_{p_2} e_2 \tag{6.61}$$
$$\hat{\sigma}_{p_1}(e_1 \hat{\bowtie}_{p_2} e_2) = e_1 \hat{\bowtie}_{p_2} \hat{\sigma}_{p_1}(e_2) \tag{6.62}$$
$$\hat{\sigma}_{p_1}(e_1 \hat{\rtimes}_{p_2} e_2) = \hat{\sigma}_{p_1}(e_1) \hat{\rtimes}_{p_2} e_2 \tag{6.63}$$
$$\hat{\sigma}_{p_1}(e_1 \hat{\mathbin{⟕}}^{g:e}_{p_2} e_2) = \hat{\sigma}_{p_1}(e_1) \hat{\mathbin{⟕}}^{g:e}_{p_2} e_2 \tag{6.64}$$
$$e_1 \hat{\times} (e_2 \hat{\times} e_3) = (e_1 \hat{\times} e_2) \hat{\times} e_3 \tag{6.65}$$
$$e_1 \hat{\bowtie}_{p_1} (e_2 \hat{\bowtie}_{p_2} e_3) = (e_1 \hat{\bowtie}_{p_1} e_2) \hat{\bowtie}_{p_2} e_3 \tag{6.66}$$
$$\hat{\sigma}_{p}(e_1 \hat{\times} e_2) = e_1 \hat{\bowtie}_{p} e_2 \tag{6.67}$$
$$e_1 \hat{<} e_2 \hat{>} = e_1 \hat{\times} e_2 \tag{6.68}$$
$$\hat{\Upsilon}_{a:f(\chi_b(e))}(\square) = \hat{\Pi}_{a:b}(f(e)) \tag{6.69}$$
$$\hat{\Upsilon}_{a:e_2}(e_1) = e_1 \hat{\times} \hat{\Upsilon}_{a:e_2}(\square) \tag{6.70}$$
$$\tag{6.71}$$

Of course, in the above equivalences the usual restrictions hold. For example, if we want to push a selection predicate into the left part of a join, it may not reference attributes of the join's right argument. In other words, $\mathcal{F}(p_1) \cap \mathcal{A}(e_2) = \emptyset$ is required. As another example, equivalence 6.70 only holds if $\mathcal{F}(e_1) \cap \mathcal{A}(e_1) = \emptyset$ In Eqv. 6.69 the function f may not alter the schema and b must be an attribute name. Please note that cross product and join are still associative in the ordered context. However, neither of them is commutative. Further, pushing selections into the second argument of a left-outer join is (in general) not possible. For strict predicates we can do better but this is beyond the scope of the paper.

6.4 Bibliography

Zaniolo [519]

6.5 Literature

- NF^2: [4, 201, 421, 714, 715, 531, 716, 532, 744]

- HAS: [116]
- Aggregates: [481, 490]
- SQL to Algebra: [866, 119]
- Calculus to Algebra: [866, 616]
- BAGs: [25]
- Algebra with control over duplicate elimination: [216]
- OO Algebra of Steenhagen et al. [800, 801, 799, 802]
- OO Algebra of Cluet and Moerkotte [186, 187].
- OO Algebra [162]
- OO Algebra [185]
- OO Algebra [370]
- OO Algebra [546]
- OO Algebra [728]
- OO Algebra [772, 771, 770, 773]
- OO Algebra [861, 862]
- OO Algebra [929]
- SAL [66]: works on lists. Intended for semistructured data. SAL can be thought of as the order-preserving counterpart of the algebra presented in [186, 187] extended to handle semistructured data. These extensions are similar to those proposed in [5, 176]
- TAX [449]: The underlying algebra's data model is based on sets of ordered labeled trees. Intended for XML.
- XML: Construction: [271, 272]
- no-name [790]: order preserving algebra for OLAP-style queries
- [360]
- Document Processing Algebras: [179, 374]
- Geo: [375]

Chapter 7

Declarative Query Representation

7.1 Calculus Representations

relational calculus originally introduced by Codd: [193, 192].

Variant for embedding in Pascal/R: [743]

calculus for complex objects: [47]

7.2 Datalog

7.3 Tableaux Representation

Tableaus have been introduced by [21, 22, 23] Tableaus are able to represent a particuar kind of queries, the so called *conjunctive queries* ([128], [701]).

Expressions containing disjunction (set union) and negation (set difference) can be represented by sets of tableaus ([727],[461]).

query processing with tables: [633]

7.4 Monoid Comprehension

[105, 267, 268]

7.5 Expressiveness

transitivity: [663]. aggregates: [490]. complex object and nested relations: [3].

7.6 Bibliography

Chapter 8

Query Equivalence, Containment, Minimization, and Factorization

This chapter is devoted to three related problems. Let q be a query and d be a database. We denote by $q(d)$ the result of evaluating the query on the database. Two queries q_1 and q_2 are by definition *equivalent* ($q_1 \equiv q_2$), if $q_1(d) = q_2(d)$ for all databases d. The problem of deciding whether two queries are equivalent is the *query equivalence* problem. We define $q_1 \subseteq q_2$ if $q_1(d) \subseteq q_2(d)$ for all databases d. The problem of deciding whether for two $q_1 \subseteq q_2$ holds is the *query containment* problem. Obviously, $q_1 \equiv q_2 \ \prec\!\succ \ q_1 \subseteq q_2 \wedge q_2 \subseteq q_1$.

The third problem is query minimization. Consider the following conjunctive query written as a rule in datalog:

$$q(X, Y) : -p(X, Y), p(X, Y).$$

Obviously, this query is equivalent to

$$q(X, Y) : -p(X, Y).$$

under set semantics. The latter query now contains fewer body literals. Query minimization now asks for an equivalent query with the least possible number of body literals. One possible approach is to successively delete a body literal until no more body literal can be deleted without violating equivalence to the original query.

The above example is also illustrative since it shows that query equivalence (and thus query containment) differs under different semantics: whereas the above two queries are equivalent under set semantics, they are not under bag semantics. To see this, consider the extensional database $\{p(a, b), p(a, b)\}$. The result of the first query contains $p(a, b)$ four times whereas the last query contains is only 2 times.

8.1 Set Semantics

8.1.1 Conjunctive Queries

A conjunctive query may contain literals where the predicate is built-in. For example, it applies comparison operators $=, <, >, \leq, \geq, \neq$. Any explicit use of equality can be

eliminated as follows. For any literal of the form $X = c$, any occurrence of X is replaced by c and the equality literal is dropped from the query clause. For any literal of the form $X = Y$, any occurrence of Y is replaced by X and the equality literal is dropped from the query clause. This procedure is not possible for the other comparison operators $<, >, \leq, \geq, \neq$ which we call *inequality opertors*. An *inequality* is any literal literal using an inequality operator.

Containment and minimization for conjunctive queries without inequalities are NP-complete Problems. First note that a tableaux directly corresponds to a conjunctive queries with all body literals having a common predicate. From that and the NP-completeness results for tableaux containment which in turn follows immediately from first order subsumption [61, 310], it follows immediately that containment of conjunctive queries is NP-complete. Chandra and Merlin proved that minimization is NP-complete [128]. The complexity of checking for equivalence of conjunctive queries is related to graph isomorphism.

EX

The procedure for checking query containment builds upon mappings from queries to queries[1]. These mappings have two different names: *homomorphism* and *containment mapping*.

Let q_1 and q_2 be the two queries for which we want to check containment. Assume the q_i are of the form

$$\begin{array}{lllll} q_1: & r_1 & :- & l_1, \ldots, l_k \\ q_2: & r_2 & :- & l'_1, \ldots, l'_m \end{array}$$

Let $\mathcal{V}(q_i)$ be the set of variables occurring in q_i, and $\mathcal{C}(q_i)$ be the set of constants occurring in q_i. Further, let h be a substitution $h : \mathcal{V}(q_2) \rightarrow (\mathcal{V}(q_1) \cup \mathcal{C}(q_1))$. We call h a *containment mapping* from q_2 to q_1, if and only if the following conditions are fulfilled:

1. $h(r_2) = r_1$ for the head literals, and
2. for all i ($1 \leq i \leq m$) there exists a j ($1 \leq j \leq k$) such that $h(l'_i) = l_j$.

The latter condition states that for each body literal l'_i in q_2 there is a body literal l_j in q_1 such that $h(l_i) = l'_i$. Note that this does not imply that h is injective or surjective.

The following theorem connects containment mappings with the containment problem:

Theorem 8.1.1 *Let q_1 and q_2 be two conjunctive queries. Then $q_1 \subseteq q_2$ if and only if there is a containment mapping h mapping q_2 to q_1.*

Consider the following example:

$$\begin{array}{lllll} q_1: & p(X_1, X_2) & :- & q(X_2, X_1), q(X_1, X_3) \\ q_2: & p(Y_1, Y_2) & :- & q(Y_2, Y_1), q(Y_3, Y_1), q(Y_1, Y_4) \end{array}$$

Consider h with $h(Y_1) = X_1$, $h(Y_2) = X_2$, $h(Y_3) = X_2$, and $h(Y_4) = X_3$. Then

$$\begin{array}{lc|ccc} l: & p(Y_1, Y_2) & q(Y_2, Y_1) & q(Y_3, Y_1) & q(Y_1, Y_4) \\ h(l): & p(X_1, X_2) & q(X_2, X_1) & q(X_2, X_1) & q(X_1, X_3) \end{array}$$

[1] In fact, Chandra and Merlin mapped *natural models* which is essentially the same.

and, hence, $q_1 \subseteq q_2$.

A query q is minimal, if it contains the minimal possible number of body literals. More formally, q is *minimal*, if for any query q' with $q \equiv q'$ the number of body literals in q' greater than or equal to the number of body literals in q. The following theorem shows that our initial thoughts on minimization are correct for conjunctive queries.

Theorem 8.1.2 *Let q be a conjunctive query. Then there is a minimal query q' equivalent to q such that q' results from q by deleting zero or more body literals.*

This suggests a simple procedure for minimizing a given query q. For every body literal check whether some containment mapping h exists such that it is subsumed by some other body literal. Note that this containment mapping must not rename head variables.

Let q and q' be two conjunctive queries. If q can be derived from q' solely by reordering body literals and renaming variables, then q and q' are called *isomorphic*. Minimal queries are unique up to some isomorphism. Obviously, minimizing conjunctive queries is also NP-complete.

Let us now come to unions of conjunctive queries. Let $Q = Q_1 \cup \ldots \cup Q_k$ and $Q' = Q'_1 \cup \ldots \cup Q'_l$ be two unions of conjunctive queries Q_i and Q'_j with a common head predicate. A containment mapping h from Q to Q' maps each Q_i to some Q'_j such that $h(Q_i) \subseteq Q'_j$. Sagiv and Yannakakis showed the following theorem [727].

Theorem 8.1.3 *Let $Q = Q_1 \cup \ldots \cup Q_k$ and $Q' = Q'_1 \cup \ldots \cup Q'_l$ be two unions of conjunctive queries Q_i and Q'_j with a common head predicate. Then $Q \subseteq Q'$ if and only if there is a containment mapping from Q to Q'.*

This theorem gives us a corollary which allows us minimizing unions of conjunctive queries by a pairwise checking of containment [727] (see also [849]).

Corollary 8.1.4 *Let $Q = Q_1 \cup \ldots \cup Q_k$ be a union of conjunctive queries Q_i with common head predicate. Then there exists a subset R of Q such that*

1. $R \equiv Q$
2. $\neg\exists\, R' \subset R \;\; R' \equiv Q$
3. *If Q_m is any equivalent to Q, then there is a containment mapping from Q_m to R but none from Q_m to any proper subset R' of R.*

This corollary implies that we can minimize a query that is a union of conjunctive queries by eliminating those conjunctive queries Q_i from it that are contained in some Q_j.

For conjunctive queries the problems of containment, equivalence, and minimization are

The problems of containment, equivalence, and minimization of conjunctive queries are most difficitult if all body literals have a common predicate p. This is quite an unrealistic assumption as typical conjunctive queries will not only self-join the same relation. A first question is thus whether there exist special cases where there are polynomial algorithms for containment checking. Another strain of work is devoted to more complex queries. As it turns out, the results become less nice and more restricted.

8.1.2 . . . with Inequalities

We now turn to conjunctive queries with inequalities in their body. For this section, we assume that the domain is totally ordered and dense. That is, for all x and y with $x < y$, there exists a z with $x < z < y$. In this context, we have the following theorem:

Theorem 8.1.5 *Assume the two conjunctive queries q_1 and q_2 are of the form*

$$\begin{array}{lrcl} q_1: & p_1 & :- & l_1, \ldots, l_k, e_1, \ldots, e_l \\ q_2: & p_2 & :- & l'_1, \ldots, l'_m, e'_1, \ldots, e'_n \end{array}$$

where p_i are the head literals, l_i and l'_i are ordinary subgoals and e_i and e'_i are inequalities. Let h be a containment mapping from q_2 to q_1 where both are restricted to their ordinary literals. If additionally for all $i = 1, \ldots, n$ we have

$$e_1, \ldots, e_l \Longrightarrow h(e'_i)$$

then $q_1 \subseteq q_2$.

This result is due to Klug [491] who used the following procedure to reason about inequalities using comparison operators in $\{=, <, \leq\}$. Given a set of inequalities L, an directed graph G is defined whose nodes are the variables and constants in L. Whenever for all $x < y$ or $x \leq y$ in L, the edge (x, y) is added to G. For all constants c and c' in L, if $c < c'$ then we add an edge (c, c'). Edges are labeled with the according comparison operator. For equality predicates, an edge in both direction is added. Given the graph G, we conclude that $x \leq y$ if there is a path from x to y and $x < y$ only if additionally at least one edge is labelled by $<$. An alternative is to use the procedure presented in Section 11.2.3 to solve the inequality inference problem. It also allows for the comparison operator $\neq$.

To see why a dense domain is important consider the domain of integers. From $1 < x < 3$ we can easily conclude that $x = 2$, a fact we can derive neither from the procedure above nor from the axioms and inference procedure presented in Section 11.2.3.

Unfortunately, the other direction of Theorem **??** is wrong as the following example shows:

$$\begin{array}{lrcl} q_1: & p(X_1, X_2) & :- & q(X_1, X_2), r(X_3, X_4), r(X_4, X_3) \\ q_2: & p(Y_1, Y_2) & :- & q(Y_1, Y_2), r(Y_3, Y_4), Y_3 \leq Y_4 \end{array}$$

Obviously, $Y_3 \leq Y_4$ cannot be implied by any non-existing inequalities from q_1. However, for q_1 to be non-empty, we must have $r(a, b)$ and $r(b, a)$ for some a and b. We also have $a \leq b$ or $b \leq a$. In the former case, we can chose $Y_3 = a$ and $Y_4 = b$ and in the latter $Y_3 = b$ and $Y_4 = a$ to satisfy $r(Y_3, Y_4)$ and $Y_3 \leq Y_4$.

Klug provides an alternative method to solve the containment problem. It builds upon *canonical models*. He then shows that if the containment test succeeds for all canonical models then and only then query containment holds [491]. Klug does not give an explicit algorithm for constructing these canonical models but these can be found in the literature []. He also gives two simple subclasses of inequality queries, where constants are allowed only on the left- or only on the right-hand side. For these subclasses if and only if holds in the above Theorem **??**.

Although the theorem is stated in terms of conjunctive queries with inequalities, it holds for any predicate p. Assume two queries of the following form:

$$\begin{array}{llll} q_1: & p_1 & :- & l_1, \ldots, l_k, P \\ q_2: & p_2 & :- & l'_1, \ldots, l'_m, P' \end{array}$$

where P and P' are arbitrary formulas. If there is a containment mapping from q_2 to q_1 where both are restricted to their ordinary literals and $P \Longrightarrow h(P')$, then $q_1 \subseteq q_2$.

8.1.3 ... with Negation

The first incarnation of negation we consider is set difference. Here, Sagiv and Yannakakis where the first to derive some results [727].

[198]

8.1.4 ... under Constraints

constraints: [459, 460, 429]

negation+constraints: [261, 262, 263, 264]

8.1.5 ... with Aggregation

[199, 200]

8.2 Bag Semantics

8.2.1 Conjunctive Queries

- definition bag-containment, bag-equivalence [216, 486]
  - characterizations [148, 436] (no proofs in [149])
  - complexity results [148]
- definition bag-set containment, bag-set equivalence [148]

8.3 Sequences

8.3.1 Path Expressions

XPath constructs and their short-hands to denote XPath sublanguages.

- branching ('[]')
- wild cards ('*')
- descendant axis ('//')
- disjunction : only binary *or* branching, ('—') (or-branching)

Otherwise XPath only contains the child axis and node name tests. These sublanguages are represented as tree patterns.

Query containment for certain subclasses:

- $XP^{[],*,//}$ is coNP-complete [589]

  Consider we have to answer $p \subseteq p'$. Then

  - if $p \in P^{[],//}$ and $p \in P^{[],*,//}$ then query containment is coNP-complete.
  - in PTIME if number of '//' is restricted by d which then gives the degree of the polynomial describing the time complexity
  - remains coNP-complete if p contains no '*' p' contains at most two '*'s
  - remains coNP-complete if p contains at most 5 branches and p' contains at most 3 branches.

- $P^{[],*}$ is in PTIME (follows from work on conjunctive acyclic queries [917], also noted by Wood [898])
- $P^{[],//}$ is in PTIME [30]
- $P^{*,//}$ is in PTIME (these are related to a fragment of regular expressions [590])
- P^{or} is in PTIME
- $P^{[],or}$ is coNP-complete
- $P^{|}$ is coNP-complete [589]
  [619] showed that $P^{[],*,//,|}$ is coNP-complete for infinite alphabets and in PSPACE for finite alphabets.
- $P^{//,|}$ is PSPACE-complete
- $P^{[],*,//}$ with variable binding and equivality tests is Π_2^p- hard [228]

A PTIME algorithm for the fragment $P^{//}$ can be found in [104]

Florescu, Levy, and Suciu showed that for a language quite similar to $P^{[],//}$ containment is NP-complete if evaluated on a graph-based data model instead of a tree-based one [275].

Calvanese et al. also consider a graph-based data model and more expressive queries [108].

[619] also contains work on languages with variable bindings with different semantics.

More result: [228]

query containment with DTDs: [899]

Schwentick gives a very good overview over complexity results for containment checking [745]. We should repeat his table here.

8.4 Minimization

minimization: [506]

8.5 Detecting common subexpressions

[273, 387, 385]

8.5.1 Simple Expressions

Simple Non-Expensive Expressions

Simple Expensive Expressions

8.5.2 Algebraic Expressions

8.6 Bibliography

In a pair of papers Aho, Sagiv, and Ullmann [19, 20] study equivalence, containment, and minimization problems for tableaux. More specifically, they introduce a restricted variant of relational expressions containing projection, natural join, and selection with predicates that only compare attributes with constants. They further assume the existence of a universal relation. That is, every relation R is the projection of the universal relation on $\mathcal{A}(R)$. Now, these restricted conjunctive queries can be expressed with tableaux. The authors tableaux equivalence, containment, and minimization problems also in the presence of functional dependences. The investigated problems are all NP-complete. Since the practical usefulness is limited we do not give the concrete results of this pair of papers.

[147, 150] contains (complexity) results for deciding query equivalence in the case of recursive and nonrecursive datalog.

View selection problem (just pointers): ...
[456, 453, 454]
[155]

- conjunctive queries: equivalence and minimization are NP-complete [128, 20] In [20] tableaux are used.
- polynomial algorithms for equivalence and minimization for simple tableaux: [20, 19]
- union of elementary differences: Π_2^p complete: remark in [727] and a pointer to the thesis of Sagiv [724].
- acyclic conjunctive queries: PTIME [917]
- equivalence (σ, $\bowtie$, π, $\cup$), equivalence (σ, $\bowtie$, π, $\cup$, $\setminus$): Π_2^P-complete [727]
- Recursive Datalog: [91]

Chapter 9

Translation and Lifting

9.1 Query Language to Calculus

9.2 Query Language to Algebra

9.3 Calculus to Algebra

9.4 Algebra to Calculus

9.5 Bibliography

Chapter 10

Functional Dependencies

10.1 Functional Dependencies

10.2 Functional Dependencies in the presence of NULL values

10.3 Deriving Functional Dependencies over algebraic operators

10.4 Bibliography

Part III

Enabling Techniques

Chapter 11

Simple Rewrites

11.1 Simple Adjustments

11.1.1 Rewriting Simple Expressions

Constant Folding

Constant subexpressions are evaluated and the result replaces the subexpression. For example an expression $1/100$ is replaced by 0.01. Other expressions like $a - 10 = 50$ can be rewritten to $a = 40$. However, the latter kind of rewrite is rarely performed by commercial systems.

Eliminate Between

A predicate of the form `Y BETWEEN X AND Z` is replaced by `X <= Y AND Y <= Z`. This step not only eliminates syntactic sugar but also enables transitivity reasoning to derive new predicates (see).

Eliminate IN

A predicate of the form `x IN (`c_1`,...,`c_n`)` is rewritten to `x = `c_1 `OR ...OR x = `c_n. This eliminates on form of the IN predicate and enables multikey index access.

Another possibility is to use a table function that produces a table with one column whose values are exactly those in the IN-list. From thereon, regular optimization takes place. This possibility is also investigated when several comparisons of a column with a constants are disjunctively connected.

Eliminating LIKE

A predicate of the form `a LIKE 'Guy'` can only be rewritten to `a = 'Guy'` if `a` is of type varchar. This is due to the different white space padding rules for LIKE and =.

Start and Stop conditions derived from LIKE predicates

A predicate of the form `a LIKE 'bla%'` gives rise to a start condition `a` $>=$ `'bla'`. Which can enable subsequent index usage. A stop predicate of the form `a` $<$ `'blb'` can also be derived. completing a range predicate for an index scan. Start and stop conditions can only be derived if there is no leading '%' in the pattern.

Pushing NOT operations down and eliminating them

NOT operations need to be pushed downwards for correctness reasons. Attention has to be paid to the IS NOT NULL and IS NULL predicates. XXX complete set of rules go into some table.

Merge AND, OR, and other associative operations

While parsing, AND and OR operations are binary. For simpler processing they are often n-ary in the internal representation. Therefor (p AND (q AND r)) is rewritten to (AND p q r).

In general, associative nested operations should be merged. Examples of other associative operations are $+$ and $*$.

Normalized Argument Order for Commutative Operations

ToDo enabling factorization, constant folding: move constants to the left Speed up evaluation of *equal*.

Eliminate - and /

$(x - y) \rightsquigarrow x + (-y)$ $x/y \rightsquigarrow x * (1/y)$

Adjust join predicates

$A = B + C$ becomes $A - C = B$ if $AandB$ are from one relation and C is from another.

Simplifying boolean expressions

The usual simplification rules for boolean expressions can be applied. For example, if a contradiction can be derived.

Eliminating ANY, SOME, and ALL

ANY and SOME operators in conjunction with a comparison operator are rewritten into disjunction of comparison predicates. For example `a` $>$ `ANY (`c_1`, `c_2`)` is rewritten to `a` $>$ c_1 `OR a` $>$ c_2. Correspondingly, an ALL operator with a constant list is rewritten into a conjunction of comparisons. For example, `a` $>$ `ALL(`c_1`, `c_2`)` is rewritten to `a` $>$ c_1 `AND a` $>$ c_2.

If a subquery occurs, then the ANY or SOME expression is rewritten to a correlated subquery in an EXIST predicate. Consider the query `a` $>$ `ANY (SELECT b FROM`

`...WHERE` p). It is rewritten to `EXISTS(SELECT ...FROM ...WHERE` p `AND a` $>$ `b)`.

Correspondingly, a subquery within an ALL operator is rewritten into a NOT EXISTS subquery. For example, `a` $>$ `(SELECT b FROM ...WHERE` p`)` is rewritten into `NOT EXISTS (SELECT b FROM ...WHERE` p `and a` $<=$ `b)`

- CASE ¡==¿ UNION

11.1.2 Normal forms for queries with disjunction

Another step of the NFST component or the first step of the rewriting component can be the transformation of boolean expressions found in *where* clauses in order to account for NULL values. Pushing **not** operators inside the boolean expression allows to use *two-valued logic* instead of *three-valued logic*. If we miss this step, we can get wrong results.

Another possible step is the subsequent transformation of the boolean expressions in **where** clauses into disjunctive normal form (DNF) or conjunctive normal form (CNF). This step is not always necessary and really depends on which plan generation approach is taken. Hence, this step could take place as late as in a preparatory step for plan generation. It is (obviously) only necessary if the query contains disjunctions. We discuss plan generation for queries with disjunctions in Section **??**.

11.2 Deriving new predicates

Given a set of conjunctive predicates, it is often possible to derive new predicates which might be helpful during query plan generation.

This section discusses ways to infer new predicates.

11.2.1 Collecting conjunctive predicates

A query predicate may not only contain the **and** connector, but also **or** or **not**.

For the inference rules in this section we need base predicates that *occur conjunctively*.

We say that a (base) predicate q occurs conjunctively in a (complex) predicate p if $p\ [q \leftarrow false]$ can be simplified to *false*. That is, if we replace every occurrence of q by *true*, the simplification rules in Figure 11.1 (Fig. **??**) simplify $p\ [q \leftarrow true]$ to *false*.

These simplification rules can be used to implement a simple member function *occursConjunctively* to determine whether a predicate occurs conjunctively in a predicate or not. Together with a member function or visitor *CollectBasePredicates*, we can compute the set of conjunctively occurring predicates. This set will form the basis for the next subsections.

11.2.2 Equality

Equality is a reflexive, symmetric and transitive binary relationship (see Fig. 11.2). Such a relation is called an *equivalence relation* Hence, a set of conjunctively occurring equality predicates implicitly partitions the set of composed terms (IUs) into disjunctive equivalence classes.

$$
\begin{aligned}
NOT\, true &\rightarrow false \\
NOT\, false &\rightarrow true \\
p\ AND\, true &\rightarrow p \\
p\ AND\, false &\rightarrow false \\
p\ OR\, true &\rightarrow true \\
p\ OR\, false &\rightarrow p
\end{aligned}
$$

Figure 11.1: Simplification rules for boolean expressions

$$
\begin{aligned}
& x = x \\
x = y & \Longrightarrow y = x \\
x = y \wedge y = z & \Longrightarrow x = z
\end{aligned}
$$

Figure 11.2: Axioms for equality

Constants: Let X be an equivalence class of equal expressions. Let Y be the set of all equality expressions that contributed to X. Then, in the query predicate we replace all expressions $x = y$ by $x = c$ and $y = c$ and subsequently eliminate redundant expressions.

$$\sigma_{x=c}(e_1 \bowtie_{x=y} e_2) \equiv \sigma_{x=c}(e_1) \times \sigma_{y=c}(e_2)$$

replace all predicates by IU=C.IU's equivalent to a constant In [205] an abstract data structure is presented that helps computing the equivalence classes fast and also allows for a fast check whether two terms (IUs) are in the same equivalence class. Since we are often interested in whether a given IU is equal to a constant - or, more specifically, equal to another IU bound to a constant -, we have to modify these algorithms such that the IU bound to a constant, if it exists, becomes the representative of its equivalence class.

For the member functions *addEqualityPredicate*, *getEqualityRepresentative* and *isInSameEqualityClass* we need an attribute _equalityRepresentative in class IU that is initialized such that it points to itself. Another member _equalityClassRank is initialized to 0. The code for the two member functions is given in Figure 11.3.

By calling *addEqualityPredicate* for all conjunctively occurring equality predicates we can build the equivalence classes.

11.2.3 Inequality

Table 11.1 gives a set of axioms used to derive new predicates from a set of conjunctively occurring inequalities S (see [849], see Fig. 11.4).

These axioms have to be applied until no more predicates can be derived. The following algorithm [849] performs this task efficiently:

1. Convert each $X < Y$ into $X \neq Y$ and $X \leq Y$.
2. Compute the transitive closure of $\leq$.
3. Apply axiom A8 until no more new predicates can be derived.
4. Reconstruct $<$ by using axiom A4.

Step 3 can be performed as follows. For any true IUs X and Y we find these IUs Z with $X \leq Z \leq Y$.

Then we check whether any two such Z's are related by $\neq$. Here, it is sufficient to check the original $\neq$ pairs in S and these derived in 1.

$$
\begin{array}{lcl}
A1 & : & X \leq X \\
A2 & : & X < Y \Rightarrow X \leq Y \\
A3 & : & X < Y \Rightarrow X \neq Y \\
A4 & : & X \leq Y \wedge X \neq Y \Rightarrow X < Y \\
A5 & : & X \neq Y \Rightarrow Y \neq X \\
A6 & : & X < Y \wedge Y < Z \Rightarrow X < Z \\
A7 & : & X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z \\
A8 & : & X \leq Z \wedge Z \leq Y \wedge X \leq W \wedge W \leq Y \wedge W \neq Z \Rightarrow X \neq Y
\end{array}
$$

Table 11.1: Axioms for inequality

11.2.4 Aggregation

Let $R_1, \ldots, R_n$ be relations or views, $A_1, \ldots, A_m$ attributes thereof, p_w and p_h predicates, and $a_1, \ldots, a_l$ expressions of the form $f_j(B_j)$ for aggregate functions f_j and attributes B_j. For a query block of the form

select $A_1, \ldots, A_k, a_1, \ldots, a_l$
from $R_1, \ldots, R_n$
where p_w
group by $A_1, \ldots, A_m$
having p_h

we consider the derivation of new predicates [536]. Obviously, the following predicates are true:

$$
\begin{array}{rcl}
\min(B) & \leq & B \\
\max(B) & \geq & B \\
\max(B) & \geq & \min(B) \\
\min(B) & \leq & \text{avg}(B) \\
\text{avg}(B) & \leq & \max(B)
\end{array}
$$

If p_w contains conjunctively a predicate $B\theta c$ for some constant c, we can further infer

	$\min(B)$	θ	c	if $\theta \in \{>, \geq\}$
	$\max(B)$	θ	c	if $\theta \in \{<, \leq\}$
	$\text{avg}(B)$	θ	c	if $\theta \in \{<, \leq, >, \geq\}$

These predicates can then be used to derive further predicates. The original and the derive predicates are usefule when the query block is embedded in another query block since we are allowed to add them to the embedding query block conjunctively (see Section 12.3).

If we know restrictions on the aggregates from some embedding query block, we might be able to add predicates to p_w. The following table contains the restrictions on an aggregate we know in the left column and the predicates we can infer in the right column:

$\max(B) \geq c$	$\rightsquigarrow$	$B \geq c$	if no other aggregation occurs
$\max(B) > c$	$\rightsquigarrow$	$B > c$	if no other aggregation occurs
$\min(B) \leq c$	$\rightsquigarrow$	$B \leq c$	if no other aggregation occurs
$\min(B) < c$	$\rightsquigarrow$	$B < c$	if no other aggregation occurs

Note that the aggregation occurring in the left column must be the only aggregation found in the query block. That is, $l = 1$ and p_h contains no aggregation other than a_1. To see why this is necessary, consider the following query

select deptNo, max(salary), min(salary)
from Employee
group by deptNo

Even if we know that `max(salary)` > 100.000, the above query block is not equivalent to

select deptNo, max(salary), min(salary)
from Employee
where salary ¿ 100.000
group by deptNo

Neither is

select deptNo, max(salary)
from Employee
group by deptNo
having avg(salary) ¿ 50.000

equivalent to

select deptNo, max(salary)
from Employee
where salary ¿ 100.000
group by deptNo
having avg(salary) ¿ 50.000

even if we know that `max(salary)` > 100.000.

11.2.5 ToDo

[565]

11.3 Eliminating Redundant Joins

11.4 Distinct Pull-Up and Push-Down

11.5 Set-Valued Attributes

In this section, we investigate the effect of query rewriting on joins involving set-valued attributes in object-relational database management systems. We show that by unnesting set-valued attributes (that are stored in an internal nested representation) prior to the actual set containment or intersection join we can improve the performance of query evaluation by an order of magnitude. By giving example query evaluation plans we show the increased possibilities for the query optimizer. This section is based on [413].

11.5.1 Introduction

The growing importance of object-relational database systems (ORDBMS) [812] has kindled a renewed interest in the efficient processing of set-valued attributes. One particular problem in this area is the joining of two relations on set-valued attributes [299, 410, 685]. Recent studies have shown that finding optimal join algorithms with set-containment predicates is very hard [107]. Nevertheless, a certain level of efficiency for joins on set-valued attributes is indispensable in practice.

Obviously, brute force evaluation via a nested-loop join is not going to be very efficient. An alternative is the introduction of special operators on the physical level of a DBMS [410, 685]. Integration of new algorithms and data structures on the physical level is problematic, however. On one hand this approach will surely result in tremendous speed-ups, but on the other hand this efficiency is purchased dearly. It is very costly to implement and integrate new algorithms robustly and reliably.

We consider an alternative approach to support set-containment and non-empty intersection join queries by compiling these join predicates away. The main idea is to unnest the set-valued attributes prior to the join. Thereby, we assume a nested internal representation [684]. This is also the underlying representation for the specific join algorithms proposed so far [410, 685]. Whereas [685] concentrates on set-containment joins, we also consider joins based on non-empty intersections. Ramasamy et al. also present a query rewrite for containment queries in [685], but on an unnested external representation, which (as shown there) exhibits very poor performance. Further, the special case of empty sets was not dealt with.

The goal of our paper is to show that by rewriting queries we can compile away the original set-containment or intersection join. As our experiments with DB2 show, our rewrite results in speed-up factors that grow linearly in the size of the input relations as compared to quadratic growth for brute-force nested-loop evaluation. The advantage of this approach—as compared to [410, 685]—is that no new join algorithms have to be added to the database system.

11.5.2 Preliminaries

In this section we give an overview of the definition of the set type. Due to the deferral of set types to SQL-4 [276], we use a syntax similar to that of Informix [1]. A possible example declaration of a table with a set-valued attribute is:

```
create table ngrams (
   setID   integer not null primary key,
   content set<char(3)>
);
```

`setID` is the key of the relation, whereas `content` stores the actual set. The components of a set can be any built-in or user-defined type. In our case we used `set<char(3)>`, because we wanted to store 3-grams (see also Section **??**). We further assume that on set-valued attributes the standard set operations and comparison operators are available.

Our rewriting method is based on unnesting the internal nested representation. The following view defining the unnested version of the above table keeps our representation more concise:

```
create view view_ngrams(setID, d, card) as (
   (select ngrams.setID, d.value, count(ngrams.content)
    from ngrams, table(unnest<char(3)>(ngrams.content)) d)
   union all
   (select ngrams.setID, NULL, 0)
    from ngrams
    where count(ngrams.content) = 0)
);
```

where `setID` identifies the corresponding set, `d` takes on the different values in `content` and `card` is the cardinality of the set. We also need `unnest<char(3)>`, a table function that returns a set in the form of a relation. As `unnest<char(3)>` returns an empty relation for an empty set, we have to consider this special case in the second subquery of the union statement, inserting a tuple containing a dummy value.

11.5.3 Query Rewrite

We are now ready to describe the queries we used to compare the nested and unnested approach. We concentrate on joins based on subset-equal and non-empty intersection predicates, because these are the difficult cases as shown in [107]. We have skipped joins involving predicates based on equality, because the efficient evaluation of these predicates is much simpler and can be done in a straightforward fashion (see [410]).

Checking Subset Equal Relation

Here is a query template for a join based on a subset-equal predicate:

[1] http://www.informix.com/documentation/

```
select n_1.setID, n_2.setID
from   ngrams n_1, ngrams n_2
where  is_subseteq(n_1.content, n_2.content) <> 0;
```

(The comparison with 0 is only needed for DB2, which does not understand the type bool.)

This query can be rewritten as follows. The basic idea is to join the unnested version of the table based on the set elements, group the tuples by their set identifiers, count the number of elements for every set identifier and compare this number with the original counts. The filter predicate `vn1.card <= vn2.card` discards some sets that cannot be in the result of the set-containment join. We also consider the case of empty sets in the second part of the query. Summarizing the rewritten query we get

```
(select vn1.setID, vn2.setID
 from   view_ngrams vn1, view_ngrams vn2
 where  vn1.d = vn2.d
 and    vn1.card <= vn2.card
 group by vn1.setID, vn1.card, vn2.setID, vn2.card
 having   count(*) = vn1.card)
union all
(select vn1.setID, vn2.setID
 from   view_ngrams vn1, view_ngrams vn2
 where  vn1.card = 0);
```

Checking Non-empty Intersection

Our query template for joins based on non-empty intersections looks as follows.

```
select n_1.setID, n_2.setID
from   ngrams n_1, ngrams n_2
where  intersects(n_1.content, n_2.content) <> 0;
```

The formulation of the unnested query is much simpler than the unnested query in Section 11.5.3. Due to our view definition, not much rewriting is necessary. We just have to take care of empty sets again, although this time in a different, simpler way.

```
select distinct vn1.setID, vn2.setID
from   view_ngrams vn1, view_ngrams vn2
where  vn1.d = vn2.d
and    vn1.card > 0;
```

11.6 Bibliography

This section is based on the investigations by Helmer and Moerkotte [413]. There, we also find a performance evaluation indicating that that the rewrites depending on the relation sizes result in speed-up factors between 5 and 50 even for moderately sized relations. Nevertheless, it is argued their, that support for set-valued attributes must be build into the DBMS. A viable alternative to the rewrites presented here is the usage

of special join algorithms for join predicates involving set-valued attributes [299, 409, 410, 567, 584, 585, 685]. Nevertheless, as has been shown by Cai, Chakaravarthy, Kaushik, and Naughton, dealing with set-valued attributes in joins theoretically (and of course practical) difficult issue [107]. Last, to efficiently support simple selection predicates on set-valued attributes, special index structures should be incorporated into the DBMS [411, 412, 414].

```
IU::addEqualityClassUnderThis(IU* lIU){
    IU*lRepresentativeThis = this -> getEqualityRepresentativeIU;
    IU*lRepresentativeArg = aIU -> getEqualityRepresentativeIU;

    lRepresentativeArg -> _equalityRepresentative =
    lRepresentativeThis;
    if(lRepresentativeArg -> _equalityClassRank >=
       lRepresentativeThis -> _equalityClassRank){
       lRepresentativeThis -> _equalityClassRank =
       lRepresentativeArg -> _equalityClass Rank + 1;
    }
}

IU::addEqualityPredicate(Compositing* p){
    IU*lLeft = p -> leftIU;
    IU*lRight = p -> rightIU;
    if (p -> isEqualityPredicateIU &&
             lLeft -> getEqualityRepresentativeIU ==
             lRight -> getEqualityRepresentativeIU){
        if(lLeft - > isBoundToConstantIU) {
           lLeft -> addEqualityClassUnderThis(lRight);
        }else
        if(lRight -> isBoundToConstantIU){
           lRight -> addEqualityClassUnderThis(lLeft),
        }else
        if (lLeft -> _equalityClassRank > lRight ->
        _equalityClassRank){
            lLeft -> addEqualityClassUnderThis(lRight)
        }else{
            lright -> addEqualityClassUnderThis(lLeft)
        }
    }
}

IU* IU:: getEqualityRepresentativeIU(){
     if (this == _equalityRepresentative){
         _equalityRepresentative = _equalityRepresentative ->
         getEqualityRepresentativeIU;
     }
     return_equalityRepresentative;
}
```

Figure 11.3:

A1			$X \leq X$
A2	$X < Y$	$\Rightarrow$	$X \leq Y$
A3	$X < Y$	$\Rightarrow$	$X \neq Y$
A4	$X \leq Y \wedge X \neq Y$	$\Rightarrow$	$X < Y$
A5	$X \neq Y$	$\Rightarrow$	$Y \neq X$
A6	$X < Y \wedge Y < Z$	$\Rightarrow$	$X < Z$
A7	$X \leq Y \wedge Y \leq Z$	$\Rightarrow$	$X \leq Z$
A8	$X \leq Z \wedge Z \leq Y \wedge X \leq W \wedge W \leq Y \wedge W \neq Z$	$\Rightarrow$	$X \neq Y$

Figure 11.4: Axioms for inequality

Chapter 12

View Merging

12.1 View Resolution

View merging can be as simple as replacing the view name in the **from** clause of a query by the view definition. We would like to call this step *view resolution*. This then results in a query with nesting in the **from** clause that can subsequently be unnested (see **??**). Consider the following example: XXX Example Other examples are given below. One must be careful not to produce variable clashes. Especially if a view is referenced several times, variables must be renamed.

12.2 Simple View Merging

Of course, these two steps can be merged into one step. The overall effect is then that the view name is replaced by all the entries in the **from** clause of the view definition and the predicate contained in the **where** clause of the view definition is conjunctively added to **where** clause of the query block whose **from** clause contained the view name. Consider the following view definition

create view

which is refenced in the following query:

View merging resuls in

However, there are a few pitfalls. This simple version of view merging can only be applied to simple select-project-join queries not containing duplicate elimination, set operations, grouping or aggregation. In these cases, complex view merging must be applied.

12.3 Predicate Move Around (Predicate pull-up and push-down)

If unnesting is not implemented or not possible, several techniques like predicate move around, semi-join techniques and magic rewriting allow the copying of predicates from one block into another block in order to reduce the number of qualifying tuples [536, 609, 610, 611, 758].

Let us briefly illustrate the main idea by means of a simple example query

```
select  e.name
from    Employee e,
        (select  d.name, d.dno
         from    Department d
         where   d.dno = e.dno and
                 d.boss.name = e.name and
                 d.boss.name like '%S') as D(dname,ddno)
where   e.dno between 1 and 10
```

which can be rewritten by predicate move around to

```
select  e.name
from    Employee e,
        (select  d.name, d.dno
         from    Department d
         where   d.dno = e.dno and
                 d.boss.name = e.name and
                 d.dno between 1 and 10 and
                 d.boss.name like '%S') as D(dname,dd no)
where   e.dno between 1 and 10 and
        e.name like '%S'
```

Predicate push-down and pull-up often occurs in conjunction with views. Let us therefore consider some examples. The following view that cannot be simply merged because it contains a **union** operator. Consider the case where there are two different employee tables that are unioned in a view.

```
create view Emp(eno, name, salary, dno) as
select  e1.eno, e1.name, e1.salary, e1.dno
from    Emp1[e1]
union all
select  e2.eno, e2.name, e2.salary, e2.dno
from    Emp2[e2]
```

Simple view merging cannot be applied to the query

```
select  e.eno, e.name
from    Emp[e]
where   e.salary > 150000
```

but view resolution with a subsequent push-down of the predicate `e.salary` $>$ `150.000` will result in

```
select  e.eno, e.name
from    ( select    e1.eno, e1.name, e1.salary, e1.dno
          from      Emp1[e1]
          where     e1.salary > 150000)
        union all select    e2.eno, e2.name, e2.salary, e2.dno
          from      Emp2[e2]
          where     e2.salary > 150000)
```

Note that we did not eliminate unneeded columns/attributes. Further note that we can now exploit possible indexes on `Emp1.salary` and `Emp2.salary`. In case **union** would have been used in the view definition, the rewritten query would also contain **union** requiring a duplicate elimination.

Here is another example where pushing a predicate down results in much more efficient plans. Given the view

```
define view EmpStat as
select  e.dno, min(e.salary) minSal, max(e.salary) maxSal, avg(e.salary) avgSal
from    Emp[e]
group by e.dno
```

the query

```
select  *
from    EmpStat[e]
where   e.dno = 10
```

can be rewritten to

```
select  e.dno, min(e.salary) minSal, max(e.salary) maxSal, avg(e.salary) avgSal
from    Emp[e]
where e.dno = 10
group by e.dno
```

which can be further simplified to

```
select  e.dno, min(e.salary) minSal, max(e.salary) maxSal, avg(e.salary) avgSal
from    Emp[e]
where e.dno = 10
```

12.4 Complex View Merging

12.4.1 Views with Distinct

XXX TODO views with distinct

12.4.2 Views with Group-By and Aggregation

Consider the following view with a group-by clause and aggregation:

```
create view AvgSalary as
select  e.dno, avg(e.salary) as avgSalary
from    Emp[e]
group by e.dno
```

The following query uses this view:

```
select  d.name, s.avgSalary)
from    Dept[d], AvgSalary[s]
where   d.location = 'Paris' and
        d.dno = s.dno
```

Using the view definition, this query can be rewritten to

```
select  d.name, avg(e.salary) as avgSalary
from    Dept[d], Emp[e]
where   d.location = 'Paris' and
        d.dno = e.dno
group by d.ROWID, d.name
```

where `d.ROWID` is a either a key-attribute like `d.dno` or a unique row identifier of the tuples in Dept. Or course, this transformation is not valid in general. The primary condition here is that we have a key-foreign key join. More specifically, `d.dno` must be the key of the `Dept` table or it must be a unique attribute.

Applying simple view resolution results in:

```
select  d.name, s.avgSalary)
from    Dept[d], (select  e.dno, avg(salary) as avgSalary
                  from    Emp[e]
                  group by e.dno) [s]
where   d.location = 'Paris' and
        d.dno = s.dno
```

This query can then be unnested using the techniques of Section **??**.

Sometimes strange results occur. Consider for example the view

```
define view EmpStat as
select  e.dno, min(e.salary) minSal, max(e.salary) maxSal, avg(e.salary) avgSal
from    Emp[e]
group by e.dno
```

If the user issues the query

select avg(minSal), **avg**(maxSal), **avg**(avgSal)
from EmpStat

view merging results in

select avg(**min**(e.salary)), **avg**(**max**(e.salary)), **avg**(**avg**(e.salary))
from Emp[e]
group by e.dno

This is perfectly o.k. You just need to think twice about it. The resulting plan will contain two group operations: XXX Plan

12.4.3 Views in IN predicates

Consider a view that contains the minimum salary for each department

create view MinSalary **as**
select e.dno, **min**(e.salary) **as** minSalary
from Emp[e]
group by e.dno

and a query asking for all those employees together with their salaries in Parisian departments earning the minimum salary:

select e.name, e.salary
from Emp[e], Dept[d]
where e.dno = d.dno **and**
d.location = 'Paris' **and**
(e.dno, e.sal) **in** MinSalary

This query can be rewritten to:

select e.name, e.salary
from Emp[e], Dept[d], Emp[e2]
where e.dno = d.dno **and**
d.location = 'Paris' **and**
e.dno = e2.dno
group by e.ROWID, d.ROWID, e.name, e.salary
having e.salary = **min**(e2.sal)

Note that the employee relation occurs twice. Avoiding to scan the employee relation twice can be done as follows:

12.4.4 Final Remarks

Not all views can be merged. If for example a `rownum` function that numbers rows in a table is used in a view definition for a result column, then the view cannot be

merged. Unmerged views will remain as nested subqueries with two alternative evaluation strategies: Either they will be evaluated as nested queries, that is for every row produced by some outer producer the view is evaluated, or the view will be materialized into a temporary table. Whatever is more efficient must be chosen by the plan generator. However, techniques for deriving additional predicates and subsequent techniques such as predicate move around (predicate pull-down, push-down) are still applicable.

12.5 Bibliography

Chapter 13

Unnesting Nested Queries

The first step in unnesting a query is *view merging*. This is simply the replacement of a view name by the view definitionThe result will always be a nested query. Unnesting a nested query that resulted from view merging is not different from unnesting any other nested query. However, due to a lack of orthogonality, the kinds of nesting arising from view merging can be different from that of "regular" nested queries. Several problems add to the complexity of query unnesting.

- Special cases like empty results lead easily to bugs like the famous count bug [483, 490, 298, 612, 613].
- If the nested query contains a grouping, special rules are needed to pull up grouping operators [146].
- Special care has to be taken for a correct duplicate treatment [530, 661, 759, 760].

The main reason for the problems was that SQL lacked expressiveness and unnesting took place at the query language level. The most important construct needed for correctly unnesting queries are outer-joins [215, 298, 480, 255, 612]. After their introduction into SQL and their usage for unnesting, reordering of outer-joins became an important topic [80, 215, 292, 612, 704]. Lately, a unifying framework for different unnesting strategies was proposed in [613].

13.1 Classification of nested queries

We start by extending the classification of nested queries given by Kim [483]. We restrict ourselves to a single nested block. Kim's classification introduces five types of nested queries one of which is not used here (Type D). The four remaining types are

Type A nested queries have a constant inner block returning single elements.

Type N nested queries have a constant inner block returning sets.

Type J nested queries have an inner block that is dependent on the outer block and return a set.

Type JA nested queries have an inner block that is dependent on the outer block and return a single element.

Obviously, the need for extending the relational classification arises from the richness of the oo model compared to the relational one and its impact on the query language. The classification we propose has three dimensions: the original one plus two that are required by the following oo characteristics. In the oo context, as opposed to the relational, (i) nested blocks may be located in any clause of a **select-from-where** query and (ii) a dependency (i.e., reference to a variable of the outer block) may be expressed in any clause of a query's inner block. We restrict the presentation to queries of type A/N/J/JA with nesting and dependency (J/JA only) in the **where** clauses.

As in the relational context, the optimization of nested queries is done by unnesting, using different kinds of joins and group operators. There are two good reasons for unnesting nested queries. The first is that the underlying evaluation plan of a nested query relies on nested loops that, as shown in [483], can be very inefficient. On the other hand, we know of good algorithms for joins and group operations (using indexes, sorting, hashing). The second reason is that algebraic operators have nice properties that can be used for further rewriting whereas nested algebraic expressions don't have them a priori.

13.2 Queries of Type A

In this case, the nested query does not have any reference to any variable defined outside its block. This is equivalent to say that the nested query does not contain any free variables. This allows its independent evaluation. It can be moved outside the query block and the result produced by the nested query can be plugged in later. This way we avoid multiple evaluation of the nested query. The treatment is independent of the place where the nesting occurred.

Consider the following simple example:

```
select x1                                  define m = max(select x2s
from   x1 in Employee                                     from   x2 in Employee)
where x1.TotSales =                                       define x2s = x2.TotSales
      max (select x2.TotSales              select x1
           from   x2 in Employee)          from   x1 in Employee
                                           where x1s = m
                                           define x1s = x1.TotSales
```

The original query is on the left-hand side. The rewritten query after NFST is shown on the right-hand side—a convention holding throughout the rest of this section. Note that the first **define** entry is independent of the second block. Hence, it is written before it. Within the implementation of the query compiler it is convenient to have an artificial outer SFWD-block to which the outer *define* clause then belongs.

Type **A** nested queries can be unnested by moving them one block up (like in the example). Sometimes, more efficient ways to to unnest these queries are possible. In the example the extent of *Student* has to be scanned twice. This can be avoided by introducing the new algebraic operator *MAX* defined as

$$MAX_f(e) \quad := \quad \{x | x \in e, \quad f(x) = max_{y \in e}(f(y))\}$$

The MAX operator can be computed in a single pass over e.

Using MAX the above query can be expressed in the algebra as

$$q \equiv MAX_{s.age}(Student[s])$$

13.3 Queries of Type N

We discuss three different kinds of predicates occurring within the outer **where** clause:

1. $f(\vec{x})$ **in select ...**
2. **not** ($f(\vec{x})$ **in select ...**)
3. $f(\vec{x})$ = ($\subseteq, \supseteq, \ldots$) **select ...**

where $\vec{x}$ represents variables of the outer block, f a function (or subquery) on these variables and $=, \subseteq, \supseteq, \ldots$ are set comparisons. Other cases are possible but not necessarily unnestable.

1. Type N queries with an **in** operator can be transformed into a semi-join by using the following equivalence[1]:

$$\sigma_{A_1 \in \chi_{A_2}(e_2)} e_1 \equiv e_1 \ltimes\!\rtimes_{A_1=A_2} e_2 \quad \text{if } A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{13.1}$$

   The first condition is obvious, the second merely stipulates that expression e_2 must be independent of expression e_1. The interest for having this equivalence and the following is obvious. As stated previously, semi-joins and anti-joins can be implemented efficiently and they allow further rewriting. Note that special care has to be taken in order to treat duplicates correctly. We use the annotations introduced earlier, to indicate the correct treatment of duplicates.

2. Also inspired by the relational type N unnesting is the following equivalence which turns a type N query with a negated **in** operator into an antijoin:

$$\sigma_{A_1 \notin \chi_{A_2}(e_2)} e_1 \equiv e_1 \rtimes_{A_1=A_2} e_2 \quad \text{if } A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{13.2}$$

We refer to [187, 799] for the third case.

These algebraic equivalences are not helpful in themselves. What is needed is a rewrite of the internal representation. We illustrate this for the first equivalence. Consider the example query

```
select  e.name
from    Employee e
where   e.dno in (select  d.dno
                  from    Department d
                  where   d.name like 'S%')
```

[1] $\mathcal{A}$ is the set of defined IUs, $\mathcal{F}$ is the set of free IUs

and its internal representation in textual form:

```
select  en
from    Employee[e] (J)
where   ed in dnos
define  en = e.name
        ed = d.dno
        dnos = (select  dd
                from    Department[d](J)
                where   dn like 'S%'
                define  dd = d.dno
                        dn = d.dname)
```

The equivalence allows us to replace the **in** predicate by a join. On the internal representation, this is reflected by adding the **from** clause of the subquery to the outer **from** clause, adding the subquery's **define** clause entries to the outer query's **define** and adding the subquery's predicate to the outer query's predicate. The result is

```
select  en
from    Employee[e] (J)
        Department[d] (SJ)
where   ed = dd and
        dn like 'S%'
define  en = e.name
        ed = e.dno
        dd = d.dno
        dn = d.dname
```

The equivalence allows us to replace the **in** predicate by a join. Note that we had to be careful about possible duplicates that would have been introduced if we added a simple join between departments and employees, since multiple department names might have started with a 'S'. Adding a **distinct** would not have helped since multiple employees might have the same name. Hence, the correct solution is to add a (SJ) annotation to the departments.

In the presence of key or unique constraints on the inner join column, no phantom duplicates can be produced. Consider the following example:

```
select  *
from    Account[a]
where   a.custno IN (select c.custno FROM Customer)
```

If `c.custno` is a key or there is a unique constraint on it, then no phantom duplicates can be introduced. Hence, it is safe to rewrite the query as

```
select  a.*
from    Account[a], Customer[c]
where   a.custno = c.custno
```

If there further exists a reference constraint, we can eliminate the join altogether. For example, if the create table statement for Account contains a **references** `Customer` **on** `custno` constraint on `custno`, then the query is equivalent to

```
select *
from   Account[a]
```

Another possibility would be to eliminate duplicates on the subqueries result. As an example consider the following query:

```
select *
from   Customer[c]
where  c.location in (select d.location
                      from   Dept[d]
```

This query can be rewritten to

```
select *
from   Customer[c],
       (select distinct d.location
       from Dept[d])
where c.location = d.location
```

Let us briefly consider queries nested in a set comparison predicate. This case does not have a counterpart in SQL. However, if we formulate the corresponding queries on a relational schema using the non-standard SQL found in [483], they would be of Type D—resolved by a division. Using standard SQL, they would require a double nesting using **EXISTS** operations. Treating Type D queries by a relational division can only treat very specific queries where the comparison predicate corresponds, in our context, to a non-strict inclusion as in the example below. The query returns the employees who have sold all the expensive products.

```
select x                                 define ExpItems = select i
from   x in Employee                                       from   i in Item
where x.SoldItems ⊇                                        where  p > 20000
      select i                                             define p = i.price
      from   i in Item                   select x
      where  i.price > 20000             from  x in Employee
                                         wherexsi ⊇ ExpItems
                                         definexsi = x.SoldItems
```

One solution to evaluate this query is to use a technique similar to that of [483] and add to our algebra an object division. If the set of expensive items is important, a well implemented division operation could do much compared to a nested loop evaluation. However, we voted against this operation for three reasons. The first reason is, as we stated before, that the division is based on a non-strict inclusion of the divider set. There are no more reasons to have this inclusion than any other set comparison ($\subseteq$,

$\supset, \ldots$). Accordingly, to be coherent, we would have to introduce one operation per set comparator (as a matter of fact, this also holds for the relational context). The second reason is that division does not have particularly nice algebraic properties that we would like to exploit. The third reason is that, since object models feature set attributes, it seems more natural to add good algorithms for dealing with selections involving set comparisons than to add new algebraic operators. Further, there already exist proposals to treat restriction predicates involving set comparison operators [801]. Thus, we prefer not to rewrite the following algebraic expression which corresponds to the translation of the above query.

$$\begin{aligned} q &\equiv \chi_x(\sigma_{xsi \supseteq ExpItems}(\chi_{xsi:x.SoldItems}(Employee[x]))) \\ ExpItems &\equiv \chi_i(\sigma_{p>20000}(\chi_{p:i.price}(Item[i]))) \end{aligned}$$

The set $ExpItems$ will be evaluated first, independently of query q. The result of its evaluation will be used by the selection $\sigma_{xsi \supseteq ExpItems}$ in the outer block. The selection itself can be evaluated using an algorithm similar to that of a relational division.

Note that there is no need to consider the negation of set comparisons, since it is possible to define for each set comparison an equivalent negated counterpart. Consider for example $\neg(e_1 \subseteq e_2)$ and the set comparison operator $\not\subseteq$ defined as $(e_1 \not\subseteq e_2) := (e_1 \setminus e_2 \neq \emptyset)$.

13.4 Queries of Type J

For Type J queries, we distinguish the same three cases as for Type N queries. Again, queries with **in** (**not in**) as the connection predicate are transformed. At the algebraic level the unnesting reads:

1.

$$\sigma_{A_1 \in \chi_{A_2}(\sigma_p(e_2))} e_1 \equiv e_1 \rtimes_{A_1=A_2 \wedge p} e_2 \quad \text{if } A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(p) \subseteq \mathcal{A}(e_1 \cup e_2), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{13.3}$$

This equivalence is similar to the one used for type N queries. It just takes into account a predicate p relying on both e_1 and e_2 (second condition).

2.

$$\sigma_{A_1 \notin \chi_{A_2}(\sigma_p(e_2))} e_1 \equiv e_1 \triangleright_{A_1=A_2} (e_2 \rtimes_p e_1) \quad \text{if } A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(p) \subseteq \mathcal{A}(e_1 \cup e_2), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \tag{13.4}$$

Type J **not in** queries cannot be translated directly using an anti-join operation: a semi-join has to be performed first.

For other cases and different unnesting possibilities see [187, 799]. The alternative unnesting strategies apply outer-joins and unary and binary grouping operations.

Now, let us consider again the case featuring a set comparison. The query below returns the employees who have sold all the items with a high-tech degree larger than the sales speciality of the employee.

```
select x                          select x
from  x in Employee               from  x in Employee
wherex.SoldItems ⊇                wherexsi ⊇ SpecialItems
      select i                    definexsi = x.SoldItems
      from  i in Item                     xs = x.speciality
      wherei.hTD > x.speciality           SpecialItems = select  i
                                                         from   i in Item
                                                         where  ihTD > xs
                                                         define ihTD = i.hTD
```

The algebraic translation of the query is splitted for reasons of clarity:

$$\begin{aligned} q &\equiv \chi_x(\sigma_{xsi\supseteq SpecialItems}(q_1)) \\ q_1 &\equiv \chi_{SpecialItems:\chi_i(\sigma_{ihTD>xs}(q_3))}(q_2) \\ q_2 &\equiv \chi_{xsi:x.SoldItems,xs:x.speciality}(Employee[x]) \\ q_3 &\equiv \chi_{ihTD:i.hTD}(Item[i]) \end{aligned}$$

The problem here is that the nested query is not constant. In order to unnest the query and avoid several costly scans over the set of items, we have to associate with each employee its corresponding set of special items. For this, we rely on the following equivalence:

$$\begin{aligned} \chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) &\equiv e_1\Gamma_{g;A_1\theta A_2;f}e_2 \\ &\quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ g \notin A_1 \cup A_2,\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset \end{aligned} \tag{13.5}$$

Applying this equivalence on q_1 results in

$$q \equiv \chi_x(\sigma_{xsi\supseteq SpecialItems}(q_2\Gamma_{SpecialItems;ihTD>xs;\chi_i}q_3))$$

The binary grouping operation can be implemented by adapting standard grouping algorithms. There is another alternative to this operation that will be given in the sequel.

Two remarks. First, note that the selection with set comparator $\supseteq$ is now evaluated between two attributes. As for type N queries, we rely on good algorithms for such selections. Second, note that the application of the equivalence did not depend on the set comparison of the predicate in the outer **where** block but on the comparison of the correlation predicate within the inner block. We will come back to this point, soon.

Eqv. 13.5 is the most general equivalence for the considered type of queries. There exist two other equivalences which deal more efficiently, using simple grouping, with two special cases. The equivalence

$$\begin{aligned} \chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) &\equiv \pi_{\overline{A_2}}(e_1 \overline{\bowtie}^{g=f(\emptyset)}_{A_1=A_2}(\Gamma_{g;A_2;f}(e_2))) \\ &\quad \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, \\ &\quad A_1 \cap A_2 = \emptyset, g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2) \end{aligned} \tag{13.6}$$

relies on the fact that the comparison of the correlation predicate is equality. The superscript $g = f(\emptyset)$ is the default value given when there is no element in the result of the

group operation which satisfies $A_1 = A_2$ for a given element of e_1. The equivalence

$$\begin{aligned}\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) \quad \equiv \quad & \pi_{A_1:A_2}(\Gamma_{g;A_2\theta;f}(e_2)) \\ & \text{if } A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset, \\ & g \notin \mathcal{A}(e_1)\cup\mathcal{A}(e_2), \\ & e_1 = \pi_{A_1:A_2}(e_2) \text{ (this implies that } A_1 = \mathcal{A}(e_1))\end{aligned} \tag{13.7}$$

relies on the fact that there exists a common range over the variables of the correlation predicate (third condition). We believe that these two cases are more common than the general case. We will show one application of Eqv. 13.6 in this section. In the next one, we will give an example using an equivalence derived from Eqv. 13.7.

Eqv. 13.5, 13.6, and 13.7 are not only useful for unnesting type J nested queries occurring within the **where** clause in a predicate utilizing set comparison. As already remarked above, applying these equivalence solely depends on the presence of a correlation predicate. Hence, they enable the derivation of alternative unnested expressions for the **in** and **not in** cases. To see this, consider $\sigma_{A\in e_2}(e_1) \equiv \sigma_{A\in B}(\chi_{B:e_2}(e_1))$. Further, as demonstrated in the next section, they play a major role in unnesting type JA nested queries. That is why they should be considered *the core* of unnesting nested queries in the oo context.

Further, alternative unnested evaluation plans avoiding the binary grouping operator can also be achieved by applying the following equivalence which produces an intermediate flat result and then groups it

$$\begin{aligned}\chi_{g:f(\sigma_{A_1'\theta A_2'}(e_2))}(e_1) \quad \equiv \quad & \Gamma_{g;A_1;f\circ\overline{\pi_{A_1}}\circ\sigma_{A_2\neq\perp_{A_2}}}(e_1 \overline{\bowtie}_{A_1'\theta A_2'} e_2) \\ & \text{if } A_i = \mathcal{A}(e_i),\ A_i' \subseteq A_i,\ g \notin A_1\cup A_2,\ \mathcal{F}(e_2)\cap A_1 = \emptyset\end{aligned} \tag{13.8}$$

where $\perp_A$ is a tuple with attributes A and null values only. Which of the Eqns. 13.5–13.8 to apply is a matter of costs.

Last, there is a variant of Eqn. 13.5 in case no selection is present:

$$\begin{aligned}\chi_{g:f(e_2)}(e_1) \quad \equiv \quad & e_1\Gamma_{g;true;f}e_2 \\ & \text{if } g \notin \mathcal{A}(e_1)\cup\mathcal{A}(e_2), \mathcal{F}(e_2)\cap\mathcal{A}(e_1)=\emptyset\end{aligned} \tag{13.9}$$

13.5 Queries of Type JA

In the relational context, the treatment of type JA queries is radically different from that of type J, N or A. It requires joins, grouping and sometimes outer-joins [215, 298] (remember, from the previous sections, that type N/J SQL queries required anti-joins and semi-joins). In the oo context, there is no difference between type J and type JA queries. The reason is that, in order to deal with set comparison, outer-joins and grouping operations have already been introduced to treat Type J queries and the resulting equivalences apply to type J and type JA queries [187, 799]. The grouping operators have been defined to allow the application of functions to the sets of grouped elements. This function might as well be an aggregate function. Thus, by applying Eqv. 13.5–13.8 aggregated type JA queries are treated in exactly the same manner as type J queries.

Note that, if the applied function of the unary Γ in Equivalence 13.6 is an aggregate function (as implied by type JA queries), then its right-hand side is equivalent to the generalized aggregation of [215].

13.6 Alternative locations

Nesting in the from clause Thus far we have only considered unnesting in the *where* clause. We briefly consider an example for unnesting in the *from* clause in order to illuminate the problems involved in correct duplicate handling.

Consider the follwing query template:

```
select  R.a, S.b
from    R, (select  s.b
            from    S s
            where   Q) as S(b)
where   P
```

This query can easily be rewritten to

```
select  R.a, S.b
from    R, S
where   P and Q
```

No problems occur with duplicates. If both queries specify **select distinct**, there is no problem either. However, in

```
select  R.a, S.b
from    R, (select  distinct s.b
            from    S s
            where   Q) as S(b)
where   P
```

duplicate removal is enforced on the inner query but not on the outer. If we just unnest is the way we did before, then both possibilities, specifying **distinct** for the outer block and not specifying it, possibly results in wrong duplicate treatment. We need the annotations introduced earlier to avoid such complications. Another alternative is to add the keys of R to the **select** clause, specify **distinct** (i.e. remove duplicates) and then project on *R.a* and *S.b* without duplicate removal.

Nesting in the select clause Although nothing forbids it, type A or N nesting rarely occurs in **select** clauses. Indeed, there is not much sense in associating a constant (set or element) to each element of a set. Should that happen, we rely on the first phase of the optimization process to factor out the constant block. Thus, it will only be evaluated once.

For type J/JA queries, nesting in the **select** clause is equivalent to nesting in the **where** clause. Remember that the application of Eqns. 13.5–13.8 did not depend on

the predicate in the outer **where** block but on the correlation predicate within the inner block. The same kind of correlation predicates is used when type J/JA nesting occurs in the **select** clause. We illustrate this with the following type JA query that associates to each department its number of employees.

```
select tuple(                                  select tuple(dept: d, emps: ce)
       dept: d,                                from   d in Department
       emps: count( select e                   define ce = count( select   e
                    from  e in Employee                           from   e in Employee
                    where e.dept=d))                              where  ed=d
from   d in Department                                            define ed=e.dept)
```

Translating this expression into the algebra yields

$$\begin{aligned} q &\equiv \chi_{[dept:d,emps:ce]}(\chi_{ce:q_1}(Department[d])) \\ q_1 &\equiv count(\chi_e(\sigma_{ed=d}(\chi_{ed:e.dept}(Employee[e])))) \end{aligned}$$

Eqv. 13.6 can be applied yielding:

$$\begin{aligned} q &\equiv \chi_{[dept:d,emps:ce]}(\pi_{\overline{ed}}(Department[d] \overline{\bowtie}^{ce=0}_{d=ed}(\Gamma_{ce;ed;count\circ\chi_e}(\chi_{ed:e.dept}Employee[e])))) \\ &\equiv \pi_{\overline{ed}}(Department[d] \overline{\bowtie}^{ce=0}_{d=ed}(\Gamma_{ce;ed;count\circ\chi_e}(\chi_{ed:e.dept}Employee[e]))) \end{aligned}$$

The zero value in the superscript $ce = 0$ corresponds to the result of the **count** function on an empty set. The transformed query can be evaluated efficiently using, for instance, a sort or an index on *Employee.dept*.

There exists one type J case where another more powerful technique can be applied: a *flatten* operation is performed on the outer block, and there is no tuple constructor within the outer block's **select** clause. As shown in [186], these queries can be optimized by pushing the *flatten* operation inside until it is applied on stored attributes; thus eliminating the nesting. For completeness, we repeat the example. The example query is

```
flatten(select                                  flatten(select  g
          select  tuple(name:c.name,age:c.age)            from    e in employee
          from    c in e.children                         define  ec = e.children
          where c.age < 18)                                       g = select  tuple(name:n,age:a)
        from e in employee)                                           from    c in ec
                                                                      where a < 18
                                                                      define n = c.name
                                                                             a = c.age )
```

The standard translation gives

$$\begin{aligned} q &\equiv flatten(\chi_g(\chi_{g:e_2}(\chi_{ec:e.children}(Emp[e])))) \\ e_2 &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c]))) \end{aligned}$$

In order to push the flatten operation inside, we have to eliminate the redundant tuple extension for the attribute g:

$$\begin{aligned} q &\equiv flatten(\chi_{e_2}(\chi_{ec:e.children}(Emp[e]))) \\ e_2 &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c]))) \end{aligned}$$

Now, we know that for linear $f : \{\tau\} \to \{\tau'\}$ that

$$flatten(\chi_f(e)) = f(flatten(e)) \tag{13.10}$$

Hence,

$$\begin{aligned} q &\equiv \chi_{[name:n,age:a]}(flatten(\chi_{e_2'})(\chi_{ec:e.children}Emp[e])) \\ e_2' &\equiv \sigma_{a<18}(\chi_{a:c.age,n:c.name}(ec[c])) \\ q &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(flatten(\chi_{e_2''}(\chi_{ec:e.children}Emp[e])))) \\ e_2'' &\equiv \chi_{age:c.age,n:c.name}(ec[c]) \\ q &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{ec[c]}(\chi_{ec:e.children}Emp[e]))))) \\ &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{e.children[c]}(Emp[e]))))) \\ &\equiv \chi_{[name:n,age:a]}(\sigma_{a<18}(\chi_{a:c.age,n:c.name}(flatten(\chi_{children[c]}(Emp))))) \\ &\equiv \sigma_{age<18}(\chi_{[name:n,age:a]}(\chi_{[a:age,n:name]}(flatten(\chi_{children}(Emp))))) \\ &\equiv \sigma_{age<18}(\chi_{[name:name,age:age]}(flatten(\chi_{children}(Emp)))) \end{aligned}$$

where redundant tuple constructions were eliminated in the last steps. Note that the flatten operation is now applied on stored data.

13.7 Different Kinds of Dependency

We distinguish three kinds of dependency: projection dependency (a reference to an outer variables occurs in the **select** clause), range dependency (. . . in the **from** clause) and predicate dependency (. . . in the **where** clause). Above, we studied queries with predicate dependency. In the sequel, we concentrate on optimization techniques required for range and projection dependencies.

Range dependency Consider the following query exhibiting a range dependency. It returns the set of employees having the same name than one of their children.

```
select x
from   x in Employee
where  x.name in select c.name
                from c in x.children
```

```
select x
from   x in Employee
where  xn in CN
define xn = x.name
       xc = x.children
       CN = select cn
            from c in xc
            define cn = c.name
```

The algebraic translation is:

$$\begin{aligned} q &\equiv \chi_x(\sigma_{xn \in CN}(\chi_{CN:nq}(\chi_{xn:x.name,xc:x.children}(Employee[x])))) \\ nq &\equiv \chi_{c.name}(xc[c]) \end{aligned}$$

In terms of unnesting, there is nothing one can do. Nevertheless, the **where** clause of the above query is equivalent to the application of the path expression *x.children.name* which passes through a set. Hence, in this case, already known optimization techniques for optimizing path expressions can be applied (see also Section 35.1).

However, there exist cases where we are able to advantagely reduce range dependencies to predicate dependencies and, hence, can unnest these queries by the above introduced techniques. The reduction relies on the existence of type extents and uses *type based rewriting* (see Section 35.1). [184, 457, 593, 595, 597]. Since it has already been described, we merely present its usage as a reduction technique useful for enabling further unnesting of range dependent subqueries. The example query is

```
select tuple (e: x.name, c: select s.customer.name    select  tuple (e: xn, c: SCN)
                           from  s in = x.sales       from    x in Employee,
                           where s.customer.city      define  xn = x.name
                                 = "Karlsruhe")               xs = x.sales
from  x in  Employee                                  SCN =   select  scn
                                                              from    s in xs
                                                              where   scc = "Karlsruhe"
                                                              define  sc = s.customer
                                                                      scn = sc.name
                                                                      scc = sc.city
```

Translation to the algebra yields

$$\begin{aligned} q &\equiv \chi_{[e:xn,c:SCN]}(\chi_{SCN:nq}(\chi_{xn:x.name,xs:x.sales}(Employee[x]))) \\ nq &\equiv \chi_{scn}(\sigma_{scc=\text{“}Karlsruhe\text{”}}(\chi_{scn:sc.name,scc:sc.city}(\chi_{sc:s.city}(xs[s])))) \end{aligned}$$

Relying on the fact that the elements of the attribute *sales* of an employee belong to the extent of the class **Sale**, the inner block of the query can be rewritten as

$$nq \equiv \chi_{sc.name}(\sigma_{scc=\text{“}Karlsruhe\text{”}}(\chi_{sc:s.customer,scc:sc.city}(\sigma_{s \in xs}(Sale[s]))))$$

Type based rewriting can be performed again using the extent of class Customer. This allows us, for instance, to use indexes on *Customer.city* and *Sale.customer* to evaluate the query. However, since our goal is unnesting and not general optimization, we do not detail on this. Concerning unnesting, it is important to note that the dependency no longer specifies the range ($xs[s]$) but now represents a predicate ($\sigma_{s \in xs}$). Herewith, the algebraic expression is of the same form as one resulting from a predicate dependency. Hence, our unnesting techniques apply.

Projection dependency Queries of this kind should be rare. If they occur, they do so in two different flavors. One nice one and one nasty one. The first occurs if, within the expression forming the **select** clause, an expression occurs whose variables all depend

on the outer block. Then, this expression has to be computed only once for each variable combination resulting from the evaluation of the outer block. Besides this expression, the evaluation of the inner block is independent of outer variables. Hence, it can be factored out resulting in a halfway efficient evaluation plan. The nasty case, where the expression contains variables from the outer and the inner block, requires in general the nested loop evaluation or cross product.

Remark Nothing restricts variables of the outer block to occur only at one place within the inner block. If there exist several dependencies, all the corresponding unnesting techniques can be applied alternatively. Hence, if for example a range and a predicate dependency occur, the latter should be used for unnesting if the range dependency cannot be resolved by type based rewriting.

13.8 Unnesting IN

requires that the nested query produces no duplicates. this can be enforced by introducing a duplicat elimination operation.

13.9 Further reading

For those who want to read more about unnesting in the relational context we recommend the classical papers [215, 298, 324, 480, 483, 490, 530, 612, 613, 704, 759, 760] as well as the newer ones [402, 760]. For unnesting in object-oriented database systems we recommend [186, 187, 799].

13.10 History

With his seminal paper Kim opened the area of unnesting nested queries in the relational context [483]. Very quickly it became clear that enormous performance gains are possible by avoiding nested-loops evaluation of nested query blocks (as proposed in [39]) by unnesting them. Almost as quickly, the subleties of unnesting became apparent. The first bugs in the original approach were detected — among them the famous count bug [490]. Retrospectively, we can detect the following problem areas:

- Special cases like empty results lead easily to bugs like the count bug [490]. These have been corrected by taking different approaches [215, 490, 480, 298, 612, 613].
- If the nested query contains a grouping, special rules are needed to pull up grouping operators [146].
- Special care has to be taken for a correct duplicate treatment [402, 530, 661, 759, 760].

The main reason for the problems was that SQL lacked expressiveness and unnesting took place at the query language level. The most important construct needed for

correctly unnesting queries are outer-joins [215, 298, 480, 255, 612]. After their introduction into SQL and their usage for unnesting, reordering of outer-joins became an important topic [80, 215, 292, 612, 704]. Lately, a unifying framework for different unnesting strategies was proposed in [613].

With the advent of object-oriented databases and their query languages, unnesting once again drew some attention from the query optimization research community [186, 187, 800, 801, 799, 802]. Different from the relational unnesting strategies which mainly performed at the (extended) SQL source level, researchers prefered to use algebras that allowed nesting. For example, a predicate of a selection operator could again contain algebraic operators. Unnesting than took place at the algebraic level. The advantage of this approach are (1) it is mainly query language independent and (2) by using algebraic equivalences, correctness proofs could be delivered.

With the arrival of XQuery [269], the field was reopened by Paparizos et al. [646]. Within their approach, unnesting takes place at the algebraic level. The underlying algebra's data model is based on sets of ordered labeled trees [449]. However, instead of using a simple equivalence, a verbal description of more than one page is used to describe detection of applicability and the according unnested plan. Since this description is verbal, it is not rigorous and indeed buggy.

13.11 Bibliography

[285] [648] [689] [44] [109]

13.12 ToDo

betrachte unnesting query:

```
select [a:a,b:b]
from a in A
     b in A.b
```

ToDo

ExtremumSpecialCases
Nested Relational Approach: [109]

Chapter 14

Optimizing Queries with Materialized Views

14.1 Conjunctive Views

14.2 Views with Grouping and Aggregation

14.3 Views with Disjunction

14.4 Bibliography

materialized view with aggregates: [797],
materialized view with disjunction: [13],
SQL Server: [326]
other: [14, 137, 138, 152, 534, 814, 846, 914] [123, 128, 154, 141, 273, 470, 518, 649, 678, 751]
some more including maintenance etc: [12, 16, 50, 86, 137, 146, 202, 373, 392] [424, 469, 533, 675, 717, 256, 797] [814, 823, 822, 938, 915] [6, 242, 243, 400]
Overview: [384]
[535]
performance eval: [84]
Stacked views: [220]
recursion: [245]
with patterns (integration): [678], [244, 246], [227]

Chapter 15

Semantic Query Rewrite

15.1 Constraints and their impact on query optimization

Using Constraints: [323, 365]

15.2 Semantic Query Rewrite

Semantic query rewrite exploits knowledge (semantic information) about the content of the object base. This knowledge is typically specified by the user. We already saw one example of user-supplied information: *inverse relationships*. As we already saw, inverse relationships can be exploited for more efficient query evaluation.

Another important piece of information is knowledge about keys. In conjunction with type inference, this information can be used during query rewrite to speed up query execution. A typical example is the following query

```
select distinct  *
from             Professor p1, Professor p2
where            p1.university.name = p2.university.name
```

By type inference, we can conclude that the expressions *p1.university* and *p2.university* are of type University. If we further knew that the name of universities are unique, that is the name is a candidate key for universities, then the query could be simplified to

```
select distinct  *
from             Professor p1, Professor p2
where            p1.university = p2.university
```

Evaluating this query does no longer necessitate accessing the universities to retrieve their *name*.

Some systems consider even more general knowledge in form of equivalences holding over user-defined functions [1, 274]. These equivalences are then used to rewrite the query. Thereby, alternatives are generated all of which are subsequently optimized.

Semantic Query Optimization: [123]

15.3 Exploiting Uniqueness in Query Optimization

[653]

15.4 Bibliography

[78] [69] [907] Foreign functions semantic rules rewrite: [141] Conjunctive Queries, Branch Minimization: [711]

Part IV

Search Space Limits and Extensions

Chapter 16

Current Search Space and Its Limits

16.1 Plans with Outer Joins, Semijoins and Antijoins

outer join reordering [284, 283, 704, 292], outer join/antijoin plan generation [688], semijoin reducer [807],

16.2 Expensive Predicates and Functions

16.3 Techniques to Reduce the Search Space

- join single row tables first
- push down SARGable predicates
- For large join queries do not apply transitivity of equality to derive new predicates and disable cross products and possibly bushy trees.

16.4 Bibliography

Chapter 17

Quantifier treatment

17.1 Pseudo-Quantifiers

Again, the clue to rewrite subqueries with a **ANY** or **ALL** predicate is to apply aggregate functions [298]. A predicate of the form

$<$ **ANY** (**select** ...
from ...
where ...)

can be transformed into the equivalent predicate

$<$ (**select** **max**(...)
from ...
where ...)

Analogously, a predicate of the form

$<$ **ALL** (**select** ...
from ...
where ...)

can be transformed into the equivalent predicate

$<$ (**select** **min**(...)
from ...
where ...)

In the above rewrite rules, the predicate $<$ can be replaced by $=$, $\leq$, etc. If the predicate is $>$ or $\geq$ then the above rules are flipped. For example, a predicate of the form $>$**ANY** becomes $>$**select min** and $>$**ALL** becomes $>$**select max**.

After the rewrites have been applied, the Type A or Type JA unnesting techniques can be applied, depending on the details of the inner query block.

17.2 Existential quantifier

Existential quantifiers can be seen as special aggregate functions and query blocks exhibiting an existential quantifier can be unnested accordingly [215]. For example, an independent existential subquery can be treated the same way as a Type A query. Nested existential quantifiers with a correlation predicate can be unnested using a semi-join. Other approaches rewrite (existential) quantifiers using the aggregate function *count* [298]. Consider the partial query pattern

```
...
where exists (select  ...
from                  ...
where                 ...)
```

It is equivalent to

```
...
where 0 > (select  count(...)
from               ...
where              ...)
```

A **not exists** like in

```
...
where not exists (select  ...
from                      ...
where                     ...)
```

is equivalent to

```
...
where 0 = (select  count(...)
from               ...
where              ...)
```

After these rewrites have been applied, the Type A or Type JA unnesting techniques can be applied, depending on the details of the inner query block.

17.3 Universal quantifier

Universal quantification is a little more complex. An overview is provided in [181]. Here is the prototypical OQL query pattern upon which our discussion of universal

Case-No.	1	2	3	4	5	6	7	8
	$p()$ $q()$	$p()$ $q(e_1)$	$p()$ $q(e_2)$	$p()$ $q(e_1, e_2)$	$p(e_1)$ $q()$	$p(e_1)$ $q(e_1)$	$p(e_1)$ $q(e_2)$	$p(e_1)$ $q(e_1, e_2)$

Case-No.	9	10	11	12	13	14	15	16
	$p(e_2)$ $q()$	$p(e_2)$ $q(e_1)$	$p(e_2)$ $q(e_2)$	$p(e_2)$ $q(e_1, e_2)$	$p(e_1, e_2)$ $q()$	$p(e_1, e_2)$ $q(e_1)$	$p(e_1, e_2)$ $q(e_2)$	$p(e_1, e_2)$ $q(e_1, e_2)$

Table 17.1: Classification Scheme According to the Variable Bindings

quantifiers nested within a query block is based:

$Q \equiv$ **select** e_1
from e_1 **in** E_1
where for all e_2 **in select** e_2
from e_2 **in** E_2
where p:
q

where p (called the *range predicate*) and q (called the *quantifier predicate*) are predicates in a subset of the variables $\{e_1, e_2\}$. This query pattern is denoted by Q.

In order to emphasize the (non-)occurrence of variables in a predicate p, we write $p(e_1, \ldots, e_n)$ if p depends on the variables $e_1, \ldots, e_n$. Using this convention, we can list all the possible cases of variable occurrence. Since both e_1 and e_2 may or may not occur in p or q, we have to consider 16 cases (see Table 17.1). All cases but 12, 15, and 16 are rather trivial. Class 12 queries can be unnested by replacing the universal quantifier by a division, set difference, anti-semijoin, or counting. Class 15 queries are treated by set difference, anti-semijoin or grouping with count aggregation. For Class 16 queries, the alternatives are set difference, anti-semijoin, and grouping with count aggregation. In all cases, special care has to be taken regarding NULL values. For details see [181].

Class 12 Let us first consider an example of a Class 12 query.

select al.name
from al **in** Airline
where for all ap **in** (**select** ap
from ap **in** Airport
where apctry = 'USA'):
ap **in** al.lounges

Define $U \equiv \pi_{ap}(\sigma_{apctry='USA'}(Airport[ap, apctry]))$. Then the three alternative algebraic expressions equivalent to this query are

- plan with division:
  if $U = \emptyset$
  then Airline[name]
  else $\mu_{ap:lounges}(Airline[name, lounges]) \div U$

- plan with set difference:

$$Airline[name] \setminus (\pi_{name}(U \rhd\!\!< _{ap \notin lounges}\ Airline[name, lounges]))$$

- plan with anti-semijoin:

$$\pi_{name}(U \overline{\rhd\!\!<}_{ap \notin lounges} Airline[name, lounges])$$

  This plan is only valid, if the projected attributes of *Airline* form a superkey.

The plan with the anti-semijoin is typically the most efficient.

In general, the plan with division is [616, 345]:

$$\mathit{if}_{\sigma_{p(e_2)}(E_2[e_2]) \neq \emptyset}(((E_1[e_1] \bowtie_{q(e_1,e_2)}\ E_2[e_2]) \div \sigma_{p(e_2)}(E_2[e_2])), E_1[e_1])$$

In case the selection $\sigma_{p(e_2)}(E_2[e_2])$ yields at least a one tuple or object, we can apply the prediate p to the divident, as in

$$\mathit{if}_{\sigma_{p(e_2)}(E_2[e_2]) \neq \emptyset}(((E_1[e_1] \bowtie_{q(e_1,e_2)}\ \sigma_{p(e_2)}(E_2[e_2])) \div \sigma_{p(e_2)}(E_2[e_2])), E_1[e_1]).$$

If the quantifier predicate $q(e_1, e_2)$ is of the form $e_2 \in e_1.\mathit{SetAttribute}$, then the join can be replaced by an unnest operator:

$$\mathit{if}_{\sigma_{p(e_2)}(E_2[e_2]) \neq \emptyset}((\mu_{e_2:\mathit{SetAttribute}}(E_1[e_1, \mathit{SetAttribute}]) \div \sigma_{p(e_2)}(E_2[e_2])), E_1[e_1])$$

Using set difference, the translation is

$$E_1[e_1] \setminus \pi_{e_1}((E_1[e_1] \times \sigma_{p(e_2)}(E_2[e_2])) \setminus (E_1[e_1] \bowtie_{q(e_1,e_2)}\ \sigma_{p(e_2)}(E_2[e_2])))$$

which can be optimized to

$$E_1[e_1] \setminus E_1[e_1] \prec\!\!> _{\neg q(e_1,e_2)} \sigma_{p(e_2)}(E_2[e_2])$$

This plan is mentioned in [799], however using a regular join instead of a semi-join.

The anti-semijoin can be employed to eliminate the set difference yielding the following plan:

$$E_1[e_1] \overline{\prec\!\!>}_{\neg q(e_1,e_2)} \sigma_{p(e_2)}(E_2[e_2])$$

This plan is in many cases the most efficient plan. However, the correctness of this plan depends on the uniqueness of e_1, i.e., the attribute(s) e_1 must be a (super) key of E_1. This is especially fulfilled in the object-oriented context if e_1 consists of or contains the object identifier.

We do not present the plans based group and count operations (see [181]).

Class 15 Here is an example query of Class 15:

```
select  al.name
from    al in Airline
where for all f in (
        select  f
        from    f in Flight
        where al = f.carrier):
        f.to.apctry != “Libya”
```

The quantifier's range formulat $\sigma_{p(e_1,e_2)}(E_2[e_2])$ is obviously not closed. It contains the free variable e_1. According to the reduction algorithm of Codd [194], the division plan is

$$(E_1[e_1] \bowtie_{\neg p(e_1,e_2) \vee q(e_2)} E_2[e_2]) \div E_2[e_2].$$

The plan with set difference is

$$E_1[e_1] \setminus \pi_{e_1}((E_1[e_1] \bowtie_{p(e_1,e_2)} E_2[e_2]) \setminus (E_1[e_1] \bowtie_{p(e_1,e_2)} \sigma_{q(e_2)}(E_2[e_2])))$$

and the most efficient plan using the anti-semijoin is

$$E_1[e_1] \overline{\rtimes}_{p(e_1,e_2)} \sigma_{\neg q(e_2)}(E_2[e_2]).$$

Class 16 Here is an example Class 16 query:

select al.name
from al **in** Airline
where for all ap **in** (
    select ap
    from ap **in** Airport
    where apctry = alctry):
    ap **in** al.lounges

The range predicate again depends on the outer level variable e_1. A valid division plan looks similar to the one for Class 15. A plan with set difference is

$$E_1[e_1] \setminus \pi_{e_1}((E_1[e_1] \bowtie_{p(e_1,e_2)} E_2[e_2]) \setminus (E_1[e_1] \bowtie_{p(e_1,e_2) \wedge q(e_1,e_2)} E_2[e_2])).$$

This plan can first be refined by replacing the seet difference of the two join expression by a semijoin resultint in

$$E_1[e_1] \setminus (E_1[e_1] \ltimes_{p(e_1,e_2) \wedge \neg q(e_1,e_2)} E_2[e_2])$$

Finally, the remaining set difference is transformed into an anti-semijoin which also covers the semijoin:

$$E_1[e_1] \overline{\rtimes}_{p(e_1,e_2) \wedge \neg q(e_1,e_2)} E_2[e_2].$$

Again, the uniqueness constraing on $E_2[e_2]$ is required for this most efficient plan to be valid.

For all discussed classes, problems with NULL values might occur. In that case, the plans have to refined [181].

17.4 Bibliography

[455] [215] [181] [687, 679]

Chapter 18

Optimizing Queries with Disjunctions

18.1 Introduction

Simple rewrites as indicated in Section **??** for IN and OR predicates that boil down to comparisons of a column with a set of constants can eliminate disjunction from the plan or push it into a multirange index access.

Another possibility that can be used for disjunctions on single columns is to use DISJOINT UNION of plans. This is a special form of UNION where conditions ensure that no phantom duplicates are produced. The DISJOINT UNION operator merely concatenates the result tables without any further overhead like duplicate elimination.

For example a predicate of the form `x = c1 or y = c2` where `x` and `y` are columns of the same table results in two predicates

1. x = c_1
2. x <> c_1 AND y = c_2

Obviously, no row can satisfy both conditions. Hence, the query `select * from R where x = c1 or y = c2` can be safely rewritten to

```
(select * from R where x = c1) DISJOINT UNION (select
* from R where x <> c1 AND y = c2
```

In case there are indexes on `x` and `y` efficient plans do exist. If they don't the table R needs to be scanned twice. This problem is avoided by using bypass plans.

DISJOIN UNIONs can also be used for join predicates. Consider the following example query: `select * from R, S where R.a = S.a OR R.b = S.a` This query can be rewritten to `(select * from R, S where R.a = S.a) DISJOINT UNION (select * from R, S where R.a <> S.a and R.b = S.b)` The general condition here is that all equality predicates have one side identical. Note that both tables are scanned and joined twice. Bypass plans will eliminate this problem.

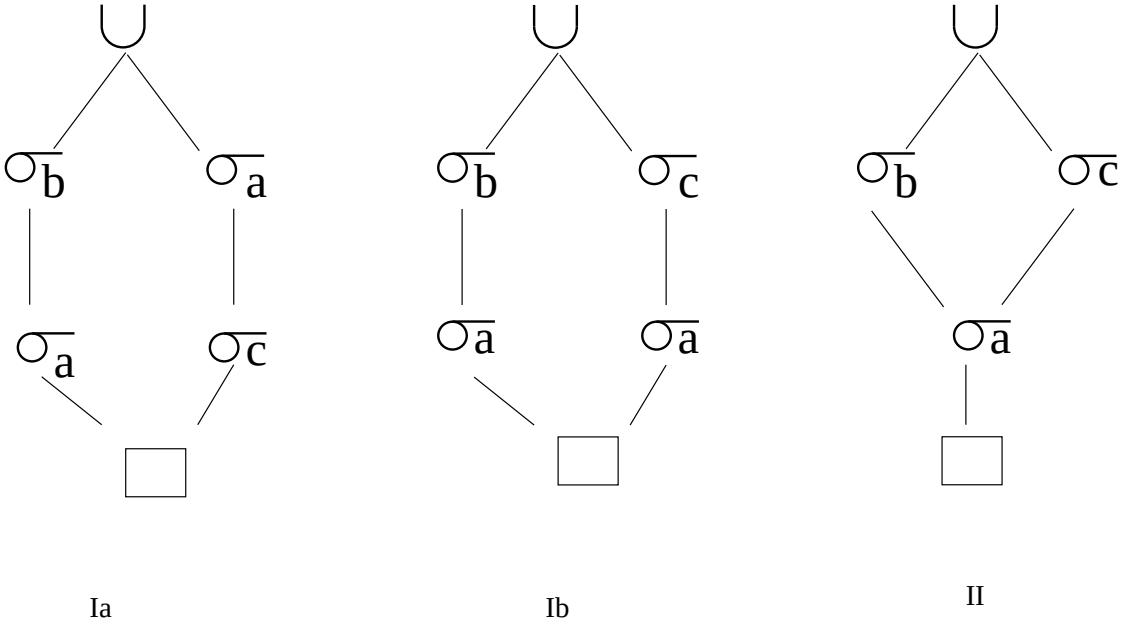
Let us consider a more complex example: `select * from R, S where R.a = S.a AND h.b IN (c1,c2)`.

XXX

18.2 Using Disjunctive or Conjunctive Normal Forms

18.3 Bypass Plans

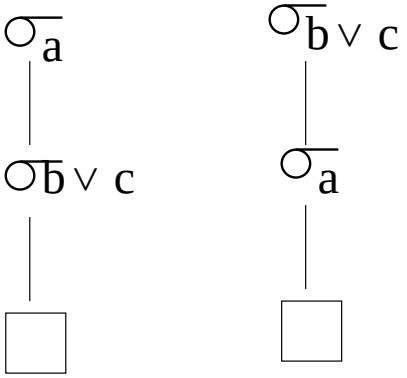
All the above approaches rely on conjunctive normal forms. However, in the presence of disjunctions, this does not necessarily yield good plans. Using a disjunctive normal form does not always solve the problem either and this approach has its own problems with duplicates. This is why bypass plans where developed [477, 804, 183]. The idea is to provide selection and join operators with two different output streams: one for qualifying tuples and one for the not qualifying tuples. We cannot go into the details of this approach and only illustrate it by means of examples. Let us first consider a query with no join and a selection predicate of the form $a \wedge (b \vee c)$. This selection predicate is already in conjunctive normal form. The disjunctive normal form is $(a \wedge b) \vee (a \wedge c)$. We first consider some DNF-based plans (Fig. 18.1). These plans generate duplicates, if a tuple qualifies for both paths. Hence, some duplicate elimination procedure is needed. Note that these duplicates have nothing to do with the duplicates generated by queries. Even if the query does not specify **distinct**, the duplicates generated must be eliminated. If there are duplicates, which is quite likely, then the condition a is evaluted twice for those tuples qualifying for both conjuncts (Plan a and b). Figure 18.2



Ia Ib II

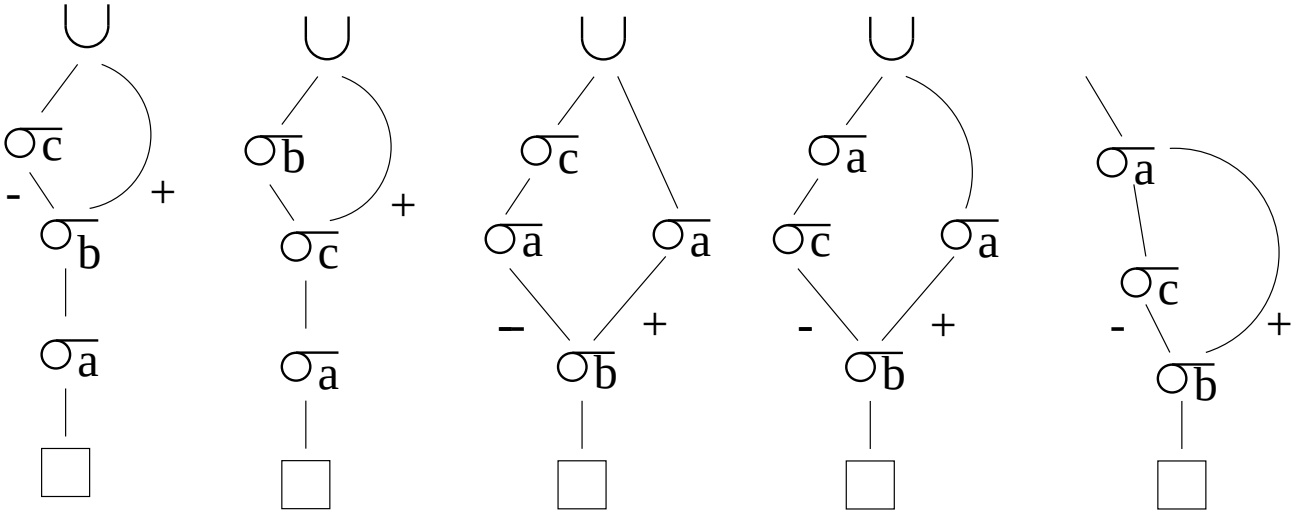
Figure 18.1: DNF plans

presents two CNF plans. CNF plans never produce duplicates. The evaluation of the boolean factors can stop as soon as some predicate evaluates to *true*. Again, some (expensive) predicates might be evaluted more than once in CNF plans. Figure 18.3 shows some bypass plans. Note the different output streams. It should be obvious, that a bypass plan can be more efficient than both a CNF or DNF plan. It is possible to extend the idea of bypass plans to join operators. However, this and the algorithm to generate bypass plans is beyond the scope of the current paper (see [477, 804, 183]).



I II

Figure 18.2: CNF plans



I II III IV V

Figure 18.3: Bypass plans

18.4 Implementation remarks

The internal representation of execution plans during plan generation typically differs from that used in Rewrite I. The reason is that many plans have to be generated and space efficiency is a major concern. As in the query representation discussed earlier, the physical algebraic operators can be organized into a hierarchy. Besides their arguments, they possibly contain backpointers to the original query representation (e.g. for predicates). Sharing is a must for plan generation. Hence, subplans are heavily shared. The plan nodes are enhanced by so-called property vectors. These contain information about the plan:

- logical information
  - the set of relations joined
  - the set of predicates applied so far
  - the set of IUs computed so far
  - order information
- physical information
  - costs
  - cardinality information

For fast processing, the first three set-valued items in the logical information block are represented as bit-vectors. However, the problem is that an upper bound on the size of these bitvectors is not reasonable. Hence, they are of variant size. It is recommendable, to have a plan node factory that generates plan nodes of different length such that the bit-vectors are included in the plan node. A special interpreter class then knows the offsets and lengths of the different bitvectors and supplies the operations needed to deal with them. This bit-vector interpreter can be attached to the plan generator's control block as indicated in Fig. 29.3.

18.5 Other plan generators/query optimizer

There are plenty of other query optimizers described in the literatur. Some of my personal favorites not mentioned so far are the Blackboard query optimzer [476], the Epoq optimizer [596, 594], the Genesis optimizer [53, 58], the Gral query optimizer [63], the Lanzelotte query optimizer [512, 513, 514], the Orion optimizer [48, 49, 482], the Postgres optimizer [458, 394, 392, 393], the Prima optimizer [398, 396], the Probe optimizer [218, 217, 629], the Straube optimizer [821, 851, 818, 819, 817, 820]. Highly recommended is a description of the DB2 query optimizer(s) [313].

Also interesting to read is the first proposal for a rule-based query optimizer called Squirel [794] and other proposals for rule-based query optimizers [280, 746, 474, 473, 566].

18.6 Bibliography

Disjunctive queries: P. Ciaccia and M. Scalas: Optimization Strategy for Relational Queries. IEEE Transaction on Software Engineering 15 (10), pp 1217-1235, 1989.

Kristofer Vorwerk, G. N. Paulley: On Implicate Discovery and Query Optimization. International Database Engineering and Applications Symposium (IDEAS'02)

Jack Minker, Rita G. Minker: Optimization of Boolean Expressions-Historical Developments. IEEE Annals of the History of Computing 2 (3), pp 227-238, 1980.

Chaudhuri: SIGMOD 03: [134]

Conjunctive Queries, Branch Minimization: [711]

Also Boolean Difference Calculus (?): [795]

Chapter 19

Grouping and Aggregation

19.1 Introduction

In general, join and grouping operations are not reorderable. Consider the following relations R and S

R	A	B
	a	5
	a	6

S	A	C
	a	7
	a	8

Joining these relations R and S results in

$R \bowtie S$	A	B	C
	a	5	7
	a	5	8
	a	6	7
	a	6	8

Applying $\Gamma_{A;count(*)}$ to R and $R \bowtie S$ yields

$\Gamma_{A;count(*)}\,(R)$	A	count $(*)$
	a	2

$\Gamma_{A;count(*)}\,(R \bowtie S)$	A	count $(*)$
	a	4

Compare this to the result of $\Gamma_{A;count(*)}(R) \bowtie S$:

$\Gamma_{A;count(*)}\,(R) \bowtie S$	A	count $(*)$	C
	a	2	7
	a	2	8

Hence $\Gamma_{A;count(*)}\,(R) \bowtie S \neq \Gamma_{A;count(*)}\,(R \bowtie S)$.

Since grouping and join operations are in general not reorderable, it is important that a query language determines the order of grouping and join operators properly. In SQL, the grouping operator is applied after the join operators of a query block.

For example, given the relations schemata

Emp (eid, name, age, salary) and
Sold (sid, eid, date, product_id, price)

and the query

select	e.eid, **sum** (s.price) as amount
from	Emp e, Sold s
where	e.eid = s.eid **and**
	s.date **between** "2000-01-01" **and** "2000-12-31"
group by	s.eid, s.name

results in the algebraic expression

$$\Pi_{e.eid,amount}\left(\Gamma_{s.eid;amount:\textbf{sum}(s.price)}\left(Emp[e] \bowtie_{e.eid=s.eid} \sigma_p\left(Sold[s]\right)\right)\right)$$

where p denotes

$$s.date \geq \text{`}2000-01-01' \wedge s.date \leq \text{`}2000-12-31'$$

Figure 20.1 (a) shows this plan graphically. Note that the grouping operator is executed last in the plan. This is the standard translation technique applied to SQL. However, Yan and Larson discovered that under certain circumstances grouping and join can be reordered [910]. In sequel work, Yan and Larson as well as Chaudhuri and Shim extended the possibilities to group before or after a join [142, 910, 911, 912, 913]. These extensions of the search space are the topic of this chapter.

Before we delve into details, let us consider an alternative plan for the above query. Here, we push down the grouping operator: The plan then becomes:

$$\Pi_{e.eid,amount}\left(Emp[e] \bowtie_{e.eid=s.eid}\left(\Gamma_{s.eid;amount:\textbf{sum}(s.price)}\left(\sigma_p\left(Sold[s]\right)\right)\right)\right)$$

This plan (see also Figure 20.1 (b)) is equivalent to the former plan. Moreover, if the grouping operator strongly reduces the cardinality of

$$\sigma_{s.date\geq...}(Sold[s])$$

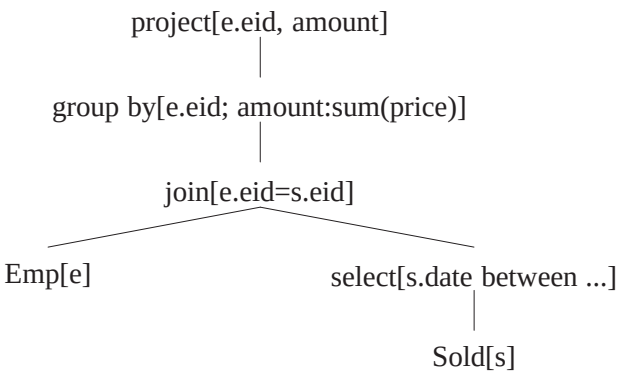
because every employee sells many items, then the latter plan might become cheaper since the join inputs are smaller than in the former plan. This motivates the search for conditions under which join and grouping operators can be reordered. Several papers discuss this reorderability and other kinds of search space extensions [142, 910, 911, 912, 913]. We will summarize their results in subsequent sections. Before that, we will take a look at aggregation functions and their properties.

The plan for the rest of the chapter is the following. First, we take a closer look at aggregate functions and their properties.

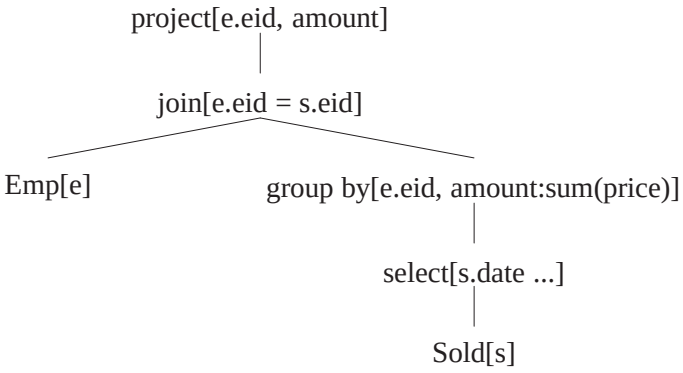
todo

19.2 Aggregate Functions

SQL and many other query languages support at least five aggregation functions. These are min, max, c$ount$, sum, and avg. In addition, SQL allows to qualify whether duplicates are removed before computing the aggregate or whether they are also considered by the aggregation function. For example, we may specify **sum(distinct** a**)** or **sum(all** a**)** for some attribute a. The term **sum(**a**)** is equivalent to **sum(all** a**)**. From



(a)



(b)

Figure 19.1: Two equivalent plans

this follows that aggregation functions can be applied to sets or bags. Other query languages (OQL and XQuery) also allow lists as arguments to aggregation functions. Additionally, OQL allows arrays. Hence, aggregation functions should be defined for any bulk type.

Most query languages provide a special null value (SQL provides **NULL**, OQL **UNKNOWN**). Typically, aggregation functions can safely ignore null values. The only exception is *count*, all input is counted independent of whether its value is null or not. If we want to count only non-null values, we could imagine an expression of the form **count(distinct not null** a**)**. Unfortunately, this is not valid SQL. However, we will make use of a variant of *count*, that does not count null values. Let us denote this function by *count*^{nn}.

Let $\mathcal{N}$ denote either a numeral data type (e.g. *integer* or *float*) or a tuple $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ where each type τ_n is a numeral data type. Further, let $\mathcal{N}$ contain a special value *NULL* denoted by $\mathcal{N}ULL$.

A *scalar* aggregation function agg is a function with signature

$$\text{agg} : \text{bulk}(()\tau) \to \mathcal{N}$$

.

A scalar aggregation function $\text{agg} : \text{bulk}(()\tau) \to \mathcal{N}$ is called *decomposable* if there exist functions

$$\begin{aligned} \text{agg}^I : \text{bulk}(()\tau) &\to \mathcal{N}' \\ \text{agg}^O : \text{bulk}(()\mathcal{N}') &\to \mathcal{N} \end{aligned}$$

with

$$\text{agg}(Z) = \text{agg}^O(\text{bulk}(()\text{agg}^I(X), \text{agg}^I(Y)))$$

for all X and Y (not empty) with $Z = X \cup_{Bulk} Y$. This condition assures that $\text{agg}(Z)$ can be computed on arbitrary subsets (-lists, -bags) of Z independently and the (partial) results can be joined to yield the correct (total) result. If the condition holds, we say that agg *is decomposable* with *inner* agg^I and *outer* agg^O. Decomposability will also be applied to vectors of aggregate functions.

A decomposable scalar aggregation function $\text{agg} : \text{bulk}(()\tau) \to \mathcal{N}$ is called *reversible* if for agg^O there exists a function $(\text{agg}^O)^{-1} : \mathcal{N}', \mathcal{N}' \to \mathcal{N}'$ with

$$\text{agg}(X) = \gamma((\text{agg}^O)^{-1}(\text{agg}^I(Z), \text{agg}^I(Y)))$$

for all X, Y, and Z with $Z = X \cup_{Bulk} Y$. This condition assures that we can compute $\text{agg}(X)$ for a subset (-list, -bag) X of Z by "subtracting" its aggregated complement Y from the "total" $\text{agg}^O(\text{agg}^I(Z))$ by using $(\text{agg}^O)^{-1}$.

The fact that scalar aggregation functions can be decomposable and reversible is the basic observation upon which the efficient evaluation of aggregation functions builds.

As an example consider the scalar aggregation $avg : \{\{()\}[a : float]) \to float$ averaging the values of the attributes a of a bag of tuples with a single attribute a. It is reversible with

$$\begin{aligned} \text{agg}^I : \{[a : float]\} &\to [sum : float, count : float] \\ \text{agg}^O : [sum : float, count : float], [sum : float, count : float] &\to [sum : float, count : float] \\ (\text{agg}^O)^{-1} : [sum : float, count : float], [sum : float, count : float] &\to [sum : float, count : float] \\ \gamma : [sum : float, count : float] &\to float \end{aligned}$$

where

$$\begin{aligned} \text{agg}^I(X) &= [sum : sum(X.a), count : |X|] \\ \text{agg}^O([sum : s_1, count : c_1], [sum : s_2, count : c_2]) &= [sum : s_1 + s_2, count : c_1 + c_2] \\ (\text{agg}^O)^{-1}([sum : s_1, count : c_1], [sum : s_2, count : c_2]) &= [sum : s_1 - s_2, count : c_1 - c_2] \\ \gamma([sum : s, count : c]) &= s/c \end{aligned}$$

$sum(X.a)$ denotes the sum of all values of attribute a of the tuples in X, and $|X|$ denotes the cardinality of X. Note that $\text{agg}^I(\emptyset) = [sum : 0, count : 0]$, and $\gamma([sum : 0, count : 0])$ is undefined as is $avg(\emptyset)$.

aggregation function!duplicated
duplicated aggregation function

Not all aggregation functions are decomposable and reversible. For instance, *min* and *max* are decomposable but not reversible. If an aggregation function is applied to a bag that has to be converted to a set, then decomposability is jeopardized for *sum* and *count*. That is, in SQL **sum(distinct)** and **count(distinct)** are not decomposable.

Let us look at the decomposition of our five aggregation functions. We can decompose them as follows:

$$\begin{aligned}
\text{min}(X \cup_{Bulk} Y) &= \text{min}(\text{min}(X), \text{min}(Y)) \\
\text{max}(X \cup_{Bulk} Y) &= \text{max}(\text{max}(X), \text{max}(Y)) \\
\text{count}(X \cup_{Bulk} Y) &= \text{sum}(\text{count}(X), \text{count}(Y)) \\
\text{sum}(X \cup_{Bulk} Y) &= \text{sum}(\text{sum}(X), \text{sum}(Y))
\end{aligned}$$

Treatment of avg is slightly more complicated, as we have already seen above. In the presence of null values, avg is defined as $\text{avg}(X) = \text{sum}(X)/\text{count}^{\text{NN}}(X)$. Hence, we can decompose it on the basis of

$$\text{avg}(X \cup_{Bulk} Y) = \text{sum}(\text{sum}(X), \text{sum}(Y))/(\text{count}^{\text{NN}}(X) + \text{count}^{\text{NN}}(Y))$$

It is also useful to classify aggregation functions as follows [912]:

- Class C Aggregate Functions: sum, count
- Class D Aggregate Functions: sum(distinct), count(distinct), min, max, avg(distinct)

Class C aggregation functions require a multiplication by the count of the inner query block. Therefore, we sometimes need to replace the vector of aggregation functions

$$F = [a_1 : \text{agg}_1(e_1), \ldots, a_n : \text{agg}_n(e_n)]$$

by the vector

$$F^{*c} = [a_1 : \text{agg}_1^{*c}(e_1), \ldots, a_n : \text{agg}_n^{*c}(e_n)]$$

$\text{agg}_i^{*c}(e_i)$ is defined as $\text{agg}_i(e_i)$ if agg_i is a class D aggregate function. If agg_i is a Class C aggregate function, we define $\text{agg}_i^{*c}(e_i)$ as $\text{agg}_i(e_i) * c$ where c is a special attribute that contains the result of some count in a subquery block. F^{*c} is called *duplicated aggregation function* of F.

19.3 Normalization and Translation

19.3.1 Grouping and Aggregation in Different Query Languages

Conversion from bags to sets must be explicitly specified within the query language. As we have seen, in SQL this is done by specifying **distinct** directly after the parenthesis following the name of the aggregation function. In OQL, the conversion function *distinct* is used. For example, the OQL query *avg(distinct(bag(1,1,2,3,3)))* return 2. Similarily for XQuery.

We now come to an essential difference between SQL and OQL/XQuery. SQL allows expressions of the form **sum(**a**)** where a is a single-valued attribute. Since aggregation functions take bulk types as arguments, this expression may seem to contain

a type error. Let us call this *false aggregates*. There are two cases to consider depending on whether the block where the aggregation function(s) occur exhibits a **group by** or not. . . .

19.3.2 Extensions to Algebra

19.3.3 Normalization

19.3.4 Translation

19.4 Lazy and eager group by

Lazy group by pulls a group operator up over a join operator [910, 911, 912, 913]. Eager group by does the opposite. This may also be called *Push-Down Grouping* and Pull-Up Grouping.
Consider the query:

select[**all** | **distinct**]	$A, \vec{F}(B)$
from	R, S
where	$p_R \wedge p_S \wedge p_{R,S}$
group by	G

with

$$G = G_R \cup G_S, G_R \subseteq \mathcal{A}(R), G_S \subseteq \mathcal{A}(S),$$

$$\mathcal{F}(p_R) \subseteq \mathcal{A}(R), \mathcal{F}(p_S) \subseteq \mathcal{A}(S)$$

$$\mathcal{F}(p_{R,S}) \subseteq \mathcal{A}(R) \cup \mathcal{A}(S)$$

$$B \subseteq \mathcal{A}(R) \qquad A = A_R \cup A_S, A_R \subseteq G_R, A_S \subseteq G_S$$

$$\text{agg}_R^I = G_R \cup \mathcal{F}(p_{R,S}) \quad \setminus \mathcal{A}(S) \quad \kappa_R \text{ key of } R$$

$$\text{agg}_S^I = G_S \cup \mathcal{F}(p_{R,S}) \quad \setminus \mathcal{A}(R) \quad \kappa_S \text{ key of } S$$

We are interested in the conditions under which the query can be rewritten into

select[**all** | **distinct**]	A, FB
from	R', S'
where	$p_{R,S}$

with $R'(\text{agg}_R^I, FB) \equiv$

select all	$\text{agg}_R^I, \vec{F}(B)$ as FB
from	R
where	p_R
group by	agg_R^I

and $S'(\text{agg}_S^I) \equiv$

select all	agg_R^I
from	S
where	p_S

The following equivalence expresses this rewrite in algebraic terms.

$$\Pi_{A,F}^{[d]}\left(\Gamma_{G;F:\vec{F}(B)}\left(\sigma_{p_R}(R) \bowtie_{p_{R,S}} \sigma_{p_S}(S)\right)\right) \equiv$$

$$\Pi_{A,F}^{[d]}\left(\Gamma_{\text{agg}_R^I;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\right) \bowtie_{p_{R,S}} \sigma_{p_S}(S)\right)$$

holds iff in $\sigma_{p_R \wedge p_S \wedge p_{R,S}}(R \times S)$ the following functional dependencies hold:
$FD_1 : G \rightarrow \text{agg}_R^I$
$FD_2 : \text{agg}_R^I, G_S \rightarrow \kappa_S$
Note that since $G_S \subseteq G$, this implies $G \rightarrow \kappa_S$.

FD_2 implies that for any group there is at most one join partner in S. Hence, each tuple in $\Gamma_{\text{agg}_R^I;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\right)$ contributes at most one row to the overall result.

FD_1 ensures that each group of the expression on the left-hand side corresponds to at most one group of the group expression on the right-hand side.

We now consider queries with a **having** clause.

In addition to the assumptions above, we have that the tables in the **from** clause can be partitioned into R and S such that R contains all aggregated columns of both the **select** and the **having** clause. We further assume that conjunctive terms in the **having** clause that do not contain aggregation functions have been moved to the **where** clause.

Let the predicate of the **having** clause have the form $H_R \wedge H_0$ where $H_R \subseteq \mathcal{A}(R)$ and $H_0 \subseteq R \cup S$ where H_0 only contains non-aggregated columns from S.

We now consider all queries of the form

select[**all** | **distinct**]	$A, \vec{F}(B)$
from	R, S
where	$p_R \wedge p_S \wedge p_{R,S}$

group by	G
having	$H_0\left(\vec{F_0}\left(B\right)\right) \wedge H_R\left(\vec{F_R}\left(B\right)\right)$

where $\vec{F_0}$ and $\vec{F_R}$ are vectors of aggregation functions on the aggregated columns B.

An alternative way to express such a query is

select[**all** | **distinct**]	G, FB
from	R', S
where	$c_S \wedge c_{R,S} \wedge H_0(F_0B)$
where	$R'\left(\text{agg}_R^I, FB, F_0B\right) \equiv$
select all	$\text{agg}_R^I, \vec{F}(B)$ as $FB, \vec{F_0}(B)$ as F_0B
from	R
where	c_R
group by	agg_R^I
having	$H_R\left(\vec{F_R}\left(B\right)\right)$

The according equivalence is [912]:

$$\Pi_{G,F}\left(\sigma_{H_R \wedge H_0}\left(\Gamma_{G;F:\vec{F}(B),F_R:\vec{F_R}(B),F_0:\vec{F_0}(B)}\left(\sigma_{p_R \wedge p_S \wedge p_{R,S}}\left(R \times S\right)\right)\right)\right)$$
$$\equiv$$
$$\Pi_{G,F}\left(\sigma_{p_{R,S} \wedge p_S \wedge H_0(F_0)}\right)\left(\Pi_{G,F,F_0}\left(\sigma_{H_R}\left(\Gamma_{G;F:\vec{F}(B),F_R:\vec{F_R}(B)F_0:\vec{F_0}(B)}(R)\right)\right) \times S\right)$$

19.5 Coalescing Grouping

In this section we introduce *coalescing grouping* which slightly generalizes *simple coalescing grouping* as introduced in [142].

We first illustrate the main idea by means of an example.
Given two relation schemes

Sales (pid, deptid, total_price)
Department (did, name, region)

the query

select	region, **sum** (total_price) as s
from	Sales, Department
where	did = deptid
group by	region

is straightforwardly translated into the following algebraic expression:

$$\Gamma_{region;s:sum(total_price)}(\text{Sales} \bowtie_{deptid=did} \text{Department})$$

Note that Equivalence **??** cannot be applied here. However, if there are many sales performed by a department, it might be worth reducing the cardinality of the left join input by introducing an additional group operator. The result is

$$\Gamma_{region;s=sum(s')}\left(\Gamma_{deptid;s':sum(total_price)}(\text{Sales}) \bowtie_{deptid=did} \text{Department}\right)$$

Note that we must keep the outer grouping.

That is, we introduced an additional group operator to reduce the cardinality of sales. This way, all subsequent joins (only one in this case) become cheaper and the additional group operator may result in a better plan.

We have the following restrictions for this section:

1. There are no NULL-values allowed for attributes occurring in the query.
2. All queries are of the form **select all**.
   That is **select distinct** is not allowed.
3. All aggregation functions agg must fulfill $\text{agg}s_1 \cup s_2 = \text{agg}\{\text{agg}(s_1), \text{agg}(s_2)\}$ for bags s_1 and s_2.
   This has two consequences:
   - Allowed are only sum, min, max. Not allowed are avg and count.
   - For any allowed aggregation function we only allow for **agg(all** ...**)**. Forbidden is **agg(distinct** ...**)**.
4. The query is a single-block conjunctive query with no **having** and no **order by** clause.

The above transformation is an application of the following equivalence, where R_1 and R_2 can be arbitrary algebraic expressions:

$$\Gamma_{G;A}\left(R_1 \bowtie_p R_2\right) \equiv \Gamma_{G;A_2}\left(\Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2\right) \tag{19.1}$$

with

$$\begin{aligned} A &= A_1 : agg_1\left(e_1\right), \ldots, A_n : agg_n\left(e_n\right) \\ A_1 &= A_1^1 : agg_1^1\left(e_1\right), \ldots, A_n^1 : agg_n^1\left(e_n\right) \\ A_2 &= A_1 : agg_1^2\left(A_1^1\right), \ldots, A_n : agg_n^2\left(A_n^2\right) \end{aligned}$$

$$G_1 = \left(\mathcal{F}\left(p\right) \cup G\right) \cap \mathcal{A}\left(R_1\right)$$

Further, the following condition must hold for all $i (1 \leq i \leq n)$:

$$agg_i\left(\bigcup_k S_k\right) = agg_i^2\left(\bigcup_k \{agg_i^1\left(S_i\right)\}\right)$$

In the above example, we had $agg_1 = agg_1^1 = agg_1^2 =$ **sum**.

We now prove the correctness of Equivalence 20.1.

Proof:
First, note that

$$R_1 \bowtie_p R_2 = \bigcup_{t_2 \in R_2} R_1 \bowtie_p \{t_2\} \tag{19.2}$$

Second, note that for a given t_2

$$\begin{aligned} \Gamma_{G;A}\left(R_1[t_1]\right) \bowtie_p \{t_2\} &= \sigma_{p(t_1 \circ t_2)}\left(\Gamma_{G;A}\left(R_1\left[t_1\right]\right)\right) \\ &= \Gamma_{G;A}\left(\sigma_{p(t_1 \circ t_2)}\left(R_1\left[t_1\right]\right)\right) \\ &= \Gamma_{G;A}\left(R_1\left[t_1\right] \bowtie_p \{t_2\}\right) \end{aligned} \tag{19.3}$$

holds where we have been a little sloppy with t_1. Applying (20.2) and (20.3) to $\Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2$, the inner part of the right-hand side of the equivalence yields:

$$\begin{aligned} \Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2 &= \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p \{t_2\} \\ &= \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}\left(R_1 \bowtie_p \{t_2\}\right) \end{aligned} \tag{19.4}$$

Call the last expression X.

Then the right-hand side of our equivalence becomes

$$\begin{aligned} \Gamma_{G;A_2}(X) = \{t \circ a_2 | t \in \Pi_G(X), a_2 = \left(A_1 : a_1^2, \ldots, A_n : a_n^2\right), \\ a_i^2 = agg_i^2\left(\{s.A_i^1 | s \in X, S|_G = t\}\right)\} \end{aligned} \tag{19.5}$$

Applying (20.2) to the left-hand side of the equivalence yields:

$$\Gamma_{G;A}\left(R_1 \bowtie_p R_2\right) = \Gamma_{G;A}\left(\bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p\left(t_1 \circ t_2\right)\}\right) \tag{19.6}$$

Abbreviate $\bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p\left(t_1 \circ t_2\right)\}$ by Y.
Applying the definition of $\Gamma_{G;A}$ yields:

$$\begin{aligned} \{t \circ a \mid t \in \Pi_G(Y), a = \left(A_1 : e_1, \ldots, A_n : e_n\right), \\ a_i = agg_i\left(\{e_i(s) | s \in Y, S|_G = t\}\right)\} \end{aligned} \tag{19.7}$$

Compare (20.5) and (20.7). Since $\Pi_G(X) = \Pi_G(Y)$, they can only differ in their values of A_i.

Hence, it suffices to prove that $a_i^2 = a_i$ for $1 \leq i \leq n$ for any given t.

$$
\begin{aligned}
a_i^2 &= agg_i^2(\{s.A_i^1 | s \in X, S|_G = t\}) \\
&= agg_i^2(\{s.A_i^1 | s \in \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}(R_1 \bowtie_p \{t_2\}), S|_G = t\}) \\
&= agg_i^2(\{s.A_i^1 | s \in \bigcup_{t_2 \in R_2} \{t_1 \circ t_2 \circ a_1 | t_1 \in \Pi_{G_1}(R_1), p(t_1 \circ t_2), \\
&\qquad\qquad a_1 = (A_1^1 : a_1^1, \ldots, A_n^1 : a_n^1) \\
&\qquad a_i^1 = agg_i^1(\{e_i(s_1 \circ t_2) | s_1 \in R_1, S_1|_{G_1 = t_1}, p(s_1, t_2)\}) \\
&\qquad S|_G = t\}\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(s_1 \circ t_2) | t_1 \in \Pi_{G_1}(R_1), p(t_1 \circ t_2), s_1 \in R_1, S_1|_{G_1} = t_1, \\
&\qquad\qquad p(s_1, t_1), t_1 \circ t_2|_G = t\})\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(s_1 \circ t_2) | s_1 \in R_1, s_1 \circ t_2|_G = t, p(s_1 \circ t_2)\})\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(t_1 \circ t_2) | t_1 \in R_1, t_1 \circ t_2|_G = t, p(t_1 \circ t_2)\})\}) \\
&= agg_i(\bigcup_{t_2 \in R_2} \{e_i(t_1 \circ t_2) | t_1 \in R_1, p(t_1 \circ t_2), t_1 \circ t_2|_G = t\}) \\
&= agg_i(\{e_i(s) | s \in \bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p(t_1 \circ t_2)\}, S|_G = t\}) \\
&= a_i
\end{aligned}
$$

Equivalence 20.1 can be used to add an additional coalescing grouping in front of any of a sequence of joins. Consider the schematic operator tree in Figure 20.2(a). It is equivalent to the one in (b), which in turn is equivalent to the one in (c) if the preconditions of Equivalence 20.1 hold. Performing a similar operation multiple times, any of the join operations can be made to be preceded by a coalescing grouping.

19.6 More Possibilities

Yan and Larson provide a whole set of possibilities to reorder grouping and join [912]. The query they consider is one of the form

select [all|distinct]	P_1, P_2, $\vec{A}_1$, $\vec{A}_2$
from	R_1, R_2
where	p_1 **and** $p_{1,2}$ **and** p_2
group by	G_1, G_2

The abbreviations used are explained below. Note that the order or grouping of entries in the different clauses is of no relevance. It was just introduced by us for convenience. An overview of all the different plans that can be produced from an initial plan is given in Figure **??**. Every of the subsequent equivalences has the initial plan as its left-hand side and one of the other plans as its right-hand side. Before giving the equivalences together with their conditions, we define some notation, some of them already used in the query above:

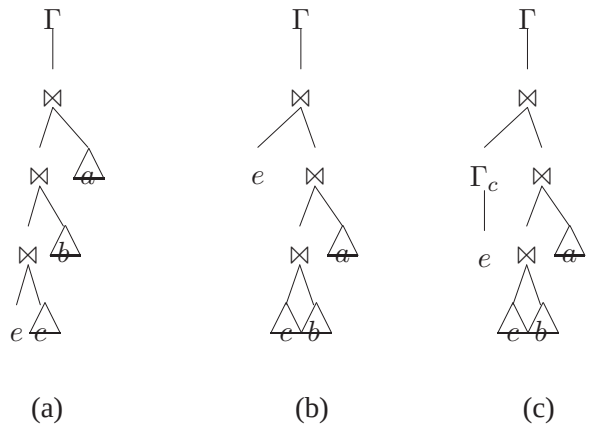


Figure 19.2: Applications of coalescing grouping

R_1 is a relation that contains aggregation columns

R_2 is a relation that may or may not contain aggregation columns

P_i are the selection columns of R_i. Define $P = P_1 \cup P_2$.

$\vec{A}_1$ is a possibly empty sequence of aggregation function calls on columns of R_1. More specifically, we assume that $\vec{A}_1$ is of the form $a_1 : \text{agg}_1(e_1), \ldots, a_k : \text{agg}_k(e_k)$ and require that $\mathcal{F}(\vec{A}_1) \subseteq \mathcal{A}(R_1)$

$\vec{A}_2$ is a possibly empty sequence of aggregation function calls on columns of R_2. More specifically, we assume that $\vec{A}_2$ is of the form $a_{k+1} : \text{agg}_1(e_{k+1}), \ldots, a_n : \text{agg}_k(e_n)$ and require that $\mathcal{F}(\vec{A}_2) \subseteq \mathcal{A}(R_2)$

$\vec{A}$ is the concatentation of $\vec{A}_1$ and $\vec{A}_2$. $\vec{A}$ may be empty.

F_i are the aggregation columns of R_1 and possibly R_2. That is, $\vec{A}^I = \mathcal{F}(\vec{A}_1)$ and $\vec{A}^O = \mathcal{F}(\vec{A}_2)$. Define $F = \vec{A}^I \cup \vec{A}^O$. Then, in case of *eager/lazy group-by*, *eager/lazy count*, and *double eager/lazy*, $F \subseteq \mathcal{A}(R_1)$.
In case of *eager/lazy group-by-count* and *eager/lazy split*, $\vec{A}^O$ may contribute to $F = \vec{A}^I \cup \vec{A}^O$.

A_i contains the columns defined by $\vec{A}_i$. More precisely, $\vec{A}_1 = a_1 : \text{agg}_1(e_1), \ldots, a_k : \text{agg}_k(e_k)$ and $\vec{A}_2 = a_1 : \text{agg}_1(e_1), \ldots, a_k : \text{agg}_k(e_k)$. resulting columns of the

application of F on AA in the first group-by when *eager group-by* is performed on the above query.

p_i is a selection predicate on columns of R_i

$p_{1,2}$ is the join predicate with columns from R_1 and R_2.

G_i are the grouping columns of R_i ($G_i \subseteq \mathcal{A}(R_i)$). Define $G = G_1 \cup G_2$.

G_i^+ are the columns of R_i participating in join and grouping, i.e. $G_i^+ := (G_i \cup \mathcal{F}(p_{1,2})) \cap \mathcal{A}(R_i)$

The query can be translated into the algebra as follows:

$$\Pi^{(D)}_{P,A}(\Gamma_{G;\vec{A}}(\sigma_{p_1 \wedge p_{1,2} \wedge p_2}(R_1 \times R_2)))$$

where the projection is duplicate eliminating if and only if the query specifies **select distinct**. P is allowed to contain more columns than those in G if these are functionally determined by G.

For the equivalences to follow, we assume that duplicates are preserved. That is, the algebraic query representation is

$$\Pi_{P,A}(\Gamma_{G;\vec{A}}(\sigma_{p_1 \wedge p_{1,2} \wedge p_2}(R_1 \times R_2)))$$

We will skip any leading Π from subsequent expressions.

19.6.1 Eager/Lazy Group-By-Count

In the following equivalence:

H_1 denotes a set of columns in R_1

c is the column produced by **count(*)** after grouping $\sigma_{p_1}(R_1)$ on H_1

A_1 the rest of the columns produced by $\vec{A}_1$ in the first group-by of table $\sigma_{p_1}(R_1)$ on H_1

$\mathbf{F}_{ua}$ the duplicated aggregation function of F_u

Further:

$$\begin{aligned} AA &= AA_d \cup AA_u \\ AA_d &= AA \cap \mathcal{A}(R_1) \\ AA_u &= AA \cap \mathcal{A}(R_2) \\ F &= A_1 \cup A_2 \end{aligned}$$

where A_1 applies to AA_d and A_2 applies to AA_u.

The expressions $E_1 :=$

$$F[AA_d, AA_u]\Pi_A[GA_d, GA_u, AA_d, AA_u]\mathcal{G}[GA_d, GA_u]\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$$

and $E_2 :=$

$$\Pi_d[GA_d, GA_u, FAA]$$
$$(F_{ua}[AA_u, CNT], F_{d2}[FAA_d])\Pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]\mathcal{G}[GA_d, GA_u]$$
$$\sigma[p_{1,2}, p_2](E_3)$$

with $E_3 :=$

$$((F_{d1}[AA_d], COUNT[])\Pi_A[H_1, GA_d^+, AA_d]\mathcal{G}[H_1]\sigma[p_1]R_1) \times R_2)$$

are equivalent if

1. A_1 contains only decomposable aggregation functions and can be decomposed into F_{d1} and F_{d2}
2. A_2 contains Class C or Class D aggregation functions
3. $H_1 \longrightarrow G_1^+$ holds in $\sigma[p_1]R_1$

Note that the equivalence assumes that duplicates are preserved (no Π_D (default in paper is Π_A which is not written) at the beginning of E_1).

The main theorem requires that the final selection columns are the same as the grouping columns, and that the final projection must not remove duplicates. This can be relaxed. The final selection columns may be a subset of the grouping columns and the final projection may remove duplicates [909].

Eager/Lazy Group-By We consider the special case where G_1 contains all the aggregation columns. In the following:

H_1 denotes a set of columns in R_1

A_1 denotes the columns produced by applying F[AA] after grouping table R_1 on H_1.

Then, the expressions $E_1 :=$

$$F[AA]\Pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u]\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$$

and $E_2 :=$

$$\vec{A}^O[FAA_d]\Pi_A[GA_d, GA_u, FAA_d]\mathcal{G}[GA_d, GA_u]\Pi_A[GA_d, GA_u, FAA_d]\sigma[p_{1,2} \wedge p_2](E_3)$$

where $E_3 :=$

$$(\vec{A}^I[AA]\Pi_A[H_1, GA_d^+, AA]\mathcal{G}[H_1]\sigma[p_1]R_1) \times R_2$$

are equivalent if

1. all aggregation functions in F[AA] are decomposable and can be decomposed into $\vec{A}^I$ and $\vec{A}^O$
2. $H_1 \longrightarrow G_1^+$ holds in $\sigma[p_1]R_1$

Proof:

- Since AA_u is empty, $F_{ua}[AA_u, CNT]$ is empty
- Deleting all terms related to AA_u in E_2 of the main theorem gives the E_2 here.

Eager/Lazy Count and Eager/Lazy Distinct We consider the special case where G_2 contains all the aggregation columns. In the following equivalence:

H_1 denotes a set of grouping (!sic!in YaLa95 for the first time) columns in R_1

CNT denotes the column produced by **count(*)** after grouping $\sigma[p_1]R_1$ on H_1

Then, the expressions $E_1 :=$

$$F[AA]\Pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u]\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$$

and $E_2 :=$

$$F_a[AA, CNT]\Pi_A[GA_d, GA_u, AA, CNT]$$
$$\mathcal{G}[GA_d, GA_u]\Pi_A[GA_d, GA_u, AA, CNT]\sigma[p_{1,2} \wedge p_2](E_3)$$

where $E_3 :=$

$$(COUNT[]\Pi_A[H_1, GA_d^+]\mathcal{G}[H_1]\sigma[p_1]R_1) \times R_2$$

are equivalent if

1. F are Class C or Class D aggregation functions
2. $H_1 \longrightarrow G_1^+$ holds in $\sigma[p_1]R_1$

COUNT[] means that we add CNT:COUNT(*) to the select list of the subquery block.

If within the equivalence F only contains Class D aggregation functions, we can simply use a **distinct** in the subquery block. We then call the transformation from E_1 to E_2 *eager distinct* and its reverse application *lazy distinct*. Note that in this case F_a is the same as F.

Proof:

- Since AA_d is empty, all of A_1, F_{d1}, and F_{d2} are empty.
- Removing all terms related to AA_d in E_2 of the main theorem gives the E_2 here.

Double Eager/Lazy In the following equivalence

H_2 denotes a set of columns in R_2

H_1 denotes a set of grouping columns in R_1

FFA denotes the columns produced by $\vec{A}^I$ in the first group-by of table $\sigma[p_1]R_1$ on H_1

CNT denotes the column produced by **count(*)** after grouping $\sigma[p_2]R_2$ on H_2

Assume that AA belongs to R_1. Then, the expressions $E_1 :=$

$$F[AA]\Pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u]\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$$

and $E_2 :=$

$$F_a[\vec{A}^O[FAA], CNT]\Pi_A[GA_d, GA_u, FAA, CNT]\mathcal{G}[GA_d, GA_u]\sigma[p_{1,2}](E_3 \times E_4)$$

where $E_3 :=$

$$(COUNT[]\Pi_A[H_2, GA_u^+]\mathcal{G}[H_2]\sigma[p_2]R_2)$$

and $E_4 :=$

$$(\vec{A}^I[AA]\Pi_A[H_1, GA_d^+, AA]\mathcal{G}[H_1]\sigma[p_1]R_1)$$

are equivalent if

1. $H_2 \longrightarrow G_2^+$ holds in $\sigma[p_1]R_2$
2. $H_1 \longrightarrow G_1^+$ holds in $\sigma[p_1]R_1$
3. all aggregation functions in F are decomposable and can be decomposed into $\vec{A}^I$ and $\vec{A}^O$
4. all aggregation functions in F are Class C or Class D and its duplicated aggregation function is F_a

If F contains only class D aggregation functions, we can use **distinct** in the subquery block of R_2. In this case F_a is the same as F.

Proof:

- eager/lazy group-by, then eager/lazy count

Double Group-By Push-Down/Pull-Up

The following equivalence shows when a top group-by can be eliminated. The equivalences $E_1 :=$

$$F[AA]\Pi_A[GA_d, GA_u, AA]\mathcal{G}[GA_d, GA_u]\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$$

and $E_2 :=$

$$\Pi_A[GA_d, GA_u, FAA * CNT]\mathcal{G}[GA_d, GA_u]\sigma[p_{1,2}](E_3 \times E_4)$$

where $E_3 :=$

$$(COUNT[]\Pi_A[H_2, GA_u^+]\mathcal{G}[H_2]\sigma[p_2]R_2)$$

and $E_4 :=$

$$(F[AA]\Pi_A[H_1, GA_d^+, AA]\mathcal{G}[H_1]\sigma[p_1]R_1)$$

are equivalent if

1. $H_2 \longrightarrow G_2^+$ holds in $\sigma_{p_1}(R_2)$
2. $H_1 \longrightarrow G_1^+$ holds in $\sigma_{p_1}(R_1)$

3. all aggregation functions in $\vec{A}$ are decomposable and can be decomposed into $\vec{A}^I$ and $\vec{A}^O$
4. all aggregation functions in F are Class C or Class D and its duplicated aggregation function is F_a
5. $G_1^+ \longrightarrow H_1$ holds in $\sigma[p_1]R_1$
6. $G_2^+ \longrightarrow H_2$ holds in $\sigma[p_2]R_2$
7. (G_1, G_2) functionally determine the join columns in $\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$

Proof in [909].

Eager/Lazy Split In the following equivalence

H_1 denotes a set of columns in R_1

H_2 denotes a set of columns in R_2

c_1 denotes the column produced by **count(*)** after grouping $\sigma[p_1]R_1$ on H_1

c_2 denotes the column produced by **count(*)** after grouping $\sigma[p_2]R_2$ on H_2

A_1 denotes the columns produced by A_1 in the first aggregation of table $\sigma[p_1]R_1$ on H_1

A_2 denotes the columns produced by A_2 in the first aggregation of table $\sigma[p_2]R_2$ on H_2

$\mathbf{F}_{da}$ denotes the duplicated aggregation function of A_1

$\mathbf{F}_{ua}$ denotes the duplicated aggregation function of A_2

Assume also that

1. $AA = AA_d \cup AA_u$ where AA_d contains only columns of R_1 and AA_u contains only columns of R_2
2. $F = A_1 \cup A_2$ where A_1 applies to AA_d and A_2 applies to AA_u

Then, the expressions $E_1 :=$

$$F[AA_d, AA_u]\Pi_A[GA_d, GA_u, AA_d, AA_u]\mathcal{G}[GA_d, GA_u]\sigma[p_1 \wedge p_{1,2} \wedge p_2](R_1 \times R_2)$$

and $E_2 :=$

$$\begin{array}{l}\Pi_d[GA_d, GA_u, FAA] \\ (F_{ua}[F_{u2}[FAA_u], CNT_1], F_{da}[F_{d2}[FAA_d], CNT_2]) \\ \Pi_A[GA_d, GA_u, FAA_u, FAA_d, CNT_1, CNT_2] \\ \mathcal{G}[GA_d, GA_u]\sigma[p_{1,2} \wedge p_2](E_3 \times E_4)\end{array}$$

where $E_3 :=$

$$(F_{d1}[AA_d], COUNT[])\Pi_A[H_1, GA_d^+, AA_d]\mathcal{G}[H_1]\sigma[p_1]R_1$$

and $E_4 :=$

$$(F_{u1}[AA_u], COUNT[])\Pi_A[H_2, GA_u^+, AA_u]\mathcal{G}[H_2]\sigma[p_2]R_2$$

are equivalent, if the following conditions hold:

1. A_1 contains only decomposable aggregation functions and can be decomposed into F_{d1} and F_{d2}

2. A_2 contains only decomposable aggregation functions and can be decomposed into F_{u1} and F_{u2}

3. A_2 and A_1 contain Class C or Class D aggregation functions

4. $H_2 \longrightarrow G_2^+$ holds in $\sigma[p_1]R_2$

5. $H_1 \longrightarrow G_1^+$ holds in $\sigma[p_1]R_1$

Proof:

- perform eager/lazy group-by-count on R_1 and then eager/lazy group-by-count on R_2

19.7 Translation into Our Notation

We define a special unary grouping operator Γ that more closely resembles grouping/aggregation in relational systems. For that reason assume that a_i ($1 \leq i \leq n$) are attribute names and e_i are expressions of the form $aggr_i(e_i')$ for some aggregation functions $aggr_i$. Denote by $\vec{A}$ the sequence $a_1 : e_1, \ldots, a_n : e_n$. We then define

$$\Gamma_{G;\vec{A}} := \{g \circ [a_1 : v_1, \ldots, a_n : v_n] | g \in \Pi_G^D(e), v_i = aggr_i(G_g)\}$$

where $G_g := \{t | t \in e, t.G = g\}$. Then

$$\Gamma_{G;\vec{A}} := \Pi^D(\chi_{[g_1:g_1,\ldots,g_k:g_k,a_1:g.a_1,\ldots,a_n:g.a_n]}(\Gamma_{g;=G;[\vec{A}]}(e)))$$

Translation Table:

Translation Table:

YaLa95	we	cmd	comment
	R	rRx	used in constructs like R_i
R_d	R_1	rRa	relation R_1
R_u	R_2	rRb	relation R_2
	P	aPx	used in constructs like P_i
SGA_d	P_1	aPa	projected columns of R_1
SGA_u	P_2	aPb	projected columns of R_2
F	$\vec{A}$	fAx	defined as $\vec{A_1} \circ fA2$
F_d	$\vec{A_1}$	fAxa	vector of aggregate functions applied to attrs of R_1
	$a_1 : \text{agg}_1(e_1), \ldots, a_k : \text{agg}_k(e_k)$	fAax	form of "
F_u	$\vec{A_2}$	fAb	vector of aggregate functions applied to attrs of R_2
	$a_{k+1} : \text{agg}_1(e_{k+1}), \ldots, a_n : \text{agg}_k(e_n)$	fAbx	form of "
AA	F	aFx	defined as $\vec{A^I} \cup aFx$
AA_d	$\vec{A^I}$	aFa	$\subseteq \mathcal{A}(()R_1)$
AA_u	$\vec{A^O}$	aFb	$\subseteq \mathcal{A}(()R_2)$
FAA	A	aAx	defined as $A_1 \cup A_2$
	A_1	aAa	columns containing aggregation result
	$< a_1, \ldots, a_k >$	aAax	form of "
	A_2	aAb	columns containing aggregation result
	$< a_{k+1}, \ldots, a_n >$	aAbx	form of "
	p	pPx	used in constructs like p_i.
C_d	p_1	pPa	selection predicate on relation R_1
C_u	p_2	pPb	selection predicate on relation R_1
C_0	$p_{1,2}$	pPj	join predicate for relations R_1 and R_2
	G	aGx	defined as $G_1 \cup G_2$
GA_d	G_1	aGa	grouping columns of R_1
GA_u	G_2	aGb	grouping columns of R_2
	G^+	aGpx	defined as $G_1^+ \cup G_2^+$
GA_d^+	G_1^+	aGpa	grouping columns plus join columns of R_1
GA_u^+	G_2^+	aGpb	grouping columns plus join columns of R_2
	H	aNx	set of attributes
NGA_d	H_1	aNa	
NGA_u	H_2	aNb	
CNT	c	c	

19.8 Aggregation of Non-Equi-Join Results

19.9 Bibliography

The main source of information for this section are the papers by Yan and Larson [910, 911, 912, 913]. These papers cover the material discussed in this section although in a different notation. An informal description of some of the ideas presented here can be found in a paper by Chaudhuri and Shim [142].

All of these papers somewhat discuss the the topic of introducing the optimal placement of Grouping and Aggregation in a plan generator. Chaudhuri and Shim devoted another paper to this important topic [144]. Duplicate removal is a special case of grouping with no aggregation taking place. Already very early on, Dayal, Goodman, and Katz observed that duplicate removal, can be pushed down beneficially [216]. This finding was confirmed by Pirahesh, Hellerstein, and Hasan [661]. Gupta, Harinarayan, and Quass [371] discusses to push down duplicate elimination into a plan

by counting the number of occurring duplicates. That is, they change the representation of the bag. After joins are performed, they reverse this represenatation change.

Pre-Aggregation: [415, 517] cardinality estimates for pre-aggregation: [416]

ToDo: [688], [712]

Aggregates: [17, 864, 143, 144, 146, 188, 189, 215] [253, 281, 357, 371, 372, 399, 480, 481] [490, 489, 598, 613, 616, 634, 769, 783, 797] [912, 936, 569]

Chapter 20

Grouping and Aggregation

20.1 Introduction

In general, join and grouping operations are not reorderable. Consider the following relations R and S

R	A	B
	a	5
	a	6

S	A	C
	a	7
	a	8

Joining these relations R and S results in

$R \bowtie S$	A	B	C
	a	5	7
	a	5	8
	a	6	7
	a	6	8

Applying $\Gamma_{A;count(*)}$ to R and $R \bowtie S$ yields

$\Gamma_{A;count(*)}\,(R)$	A	count $(*)$
	a	2

$\Gamma_{A;count(*)}\,(R \bowtie S)$	A	count $(*)$
	a	4

Compare this to the result of $\Gamma_{A;count(*)}(R) \bowtie S$:

$\Gamma_{A;count(*)}\,(R) \bowtie S$	A	count $(*)$	C
	a	2	7
	a	2	8

Hence $\Gamma_{A;count(*)}\,(R) \bowtie S \neq \Gamma_{A;count(*)}\,(R \bowtie S)$.

Since grouping and join operations are in general not reorderable, it is important that a query language determines the order of grouping and join operators properly. In SQL, the grouping operator is applied after the join operators of a query block.

For example, given the relations schemata

Emp (eid, name, age, salary) and
Sold (sid, eid, date, product_id, price)

and the query

select	e.eid, **sum** (s.price) as amount
from	Emp e, Sold s
where	e.eid = s.eid **and**
	s.date **between** "2000-01-01" **and** "2000-12-31"
group by	s.eid, s.name

results in the algebraic expression

$$\Pi_{e.eid,amount}\left(\Gamma_{s.eid;amount:\textbf{sum}(s.price)}\left(Emp[e] \bowtie_{e.eid=s.eid} \sigma_p\left(Sold[s]\right)\right)\right)$$

where p denotes

$$s.date \geq \text{`}2000-01-01' \wedge s.date \leq \text{`}2000-12-31'$$

Figure 20.1 (a) shows this plan graphically. Note that the grouping operator is executed last in the plan.

Now consider the plan where we push the grouping operator down:

$$\Pi_{e.eid,amount}\left(Emp[e] \bowtie_{e.eid=s.eid}\left(\Gamma_{s.eid;amount:\textbf{sum}(s.price)}\left(\sigma_p\left(Sold[s]\right)\right)\right)\right)$$

This plan (see also Figure 20.1 (b)) is equivalent to the former plan. Moreover, if the grouping operator strongly reduces the cardinality of

$$\sigma_{s.date\geq...}(Sold[s])$$

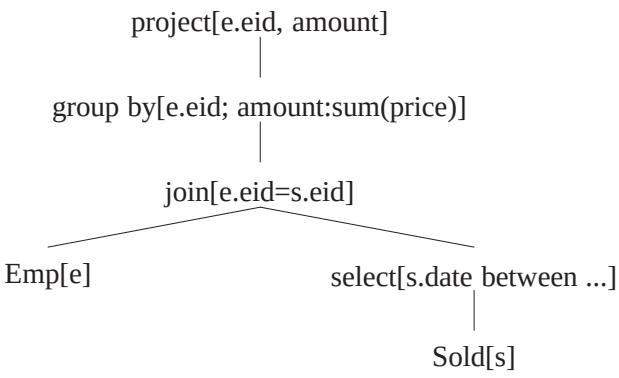
because every employee sells many items, then the latter plan might become cheaper since the join inputs are smaller than in the former plan. This motivates the search for conditions under which join and grouping operators can be reordered. Several papers discuss this reorderability [142, 910, 911, 912, 913]. We will summarize their results in subsequent sections.

20.2 Lazy and eager group by

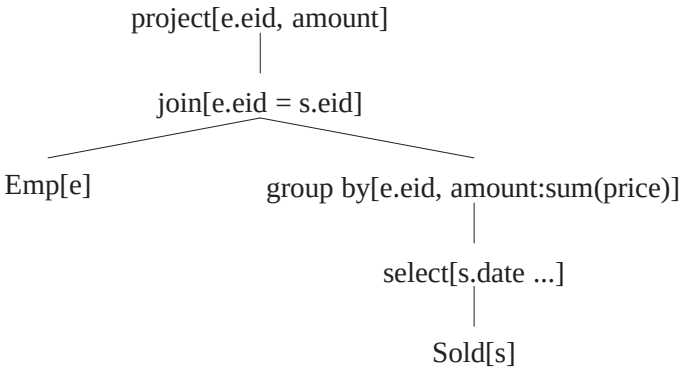
Lazy group by pulls a group operator up over a join operator [910, 911, 912, 913]. Eager group by does the opposite. This may also be called *Push-Down Grouping* and Pull-Up Grouping.

Consider the query:

select[**all** | **distinct**]	$A, \vec{F}(B)$
from	R, S
where	$p_R \wedge p_S \wedge p_{R,S}$
group by	G



(a)



(b)

Figure 20.1: Two equivalent plans

with

$$G = G_R \cup G_S, G_R \subseteq \mathcal{A}(R), G_S \subseteq \mathcal{A}(S),$$

$$\mathcal{F}(p_R) \subseteq \mathcal{A}(R), \mathcal{F}(p_S) \subseteq \mathcal{A}(S)$$

$$\mathcal{F}(p_{R,S}) \subseteq \mathcal{A}(R) \cup \mathcal{A}(S)$$

$$B \subseteq \mathcal{A}(R) \qquad A = A_R \cup A_S, A_R \subseteq G_R, A_S \subseteq G_S$$

$$\alpha_R = G_R \cup \mathcal{F}(p_{R,S}) \quad \setminus \mathcal{A}(S) \quad \kappa_R \text{ key of } R$$

$$\alpha_S = G_S \cup \mathcal{F}(p_{R,S}) \quad \setminus \mathcal{A}(R) \quad \kappa_S \text{ key of } S$$

We are interested in the conditions under which the query can be rewritten into

select[**all** | **distinct**]	A, FB
from	R', S'
where	$p_{R,S}$
with	$R'(\alpha_R, FB) \equiv$
	select all $\alpha_R, \vec{F}(B)$ as FB
	from R
	where p_R
	group by α_R
and	$S'(\alpha_S) \equiv$
	select all α_R
	from S
	where p_S

The following equivalence expresses this rewrite in algebraic terms.

$$\Pi^{[d]}_{A,F}\left(\Gamma_{G;F:\vec{F}(B)}\left(\sigma_{p_R}(R) \bowtie_{p_{R,S}} \sigma_{p_S}(S)\right)\right) \equiv$$

$$\Pi^{[d]}_{A,F}\left(\Gamma_{\alpha_R;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\right) \bowtie_{p_{R,S}} \sigma_{p_S}(S)\right)$$

holds iff in $\sigma_{p_R \wedge p_S \wedge p_{R,S}}(R \times S)$ the following functional dependencies hold:
$FD_1 : G \rightarrow \alpha_R$
$FD_2 : \alpha_R, G_S \rightarrow \kappa_S$
Note that since $G_S \subseteq G$, this implies $G \rightarrow \kappa_S$.

FD_2 implies that for any group there is at most one join partner in S. Hence, each tuple in $\Gamma_{\alpha_R;F:\vec{F}(B)}\left(\sigma_{p_R}(R)\right)$ contributes at most one row to the overall result.

FD_1 ensures that each group of the expression on the left-hand side corresponds to at most one group of the group expression on the right-hand side.

We now consider queries with a **having** clause.

In addition to the assumptions above, we have that the tables in the **from** clause can be partitioned into R and S such that R contains all aggregated columns of both the **select** and the **having** clause. We further assume that conjunctive terms in the **having** clause that do not contain aggregate functions have been moved to the **where** clause.

Let the predicate of the **having** clause have the form $H_R \wedge H_0$ where $H_R \subseteq \mathcal{A}(R)$ and $H_0 \subseteq R \cup S$ where H_0 only contains non-aggregated columns from S.

We now consider all queries of the form

select[**all** | **distinct**]	$A, \vec{F}(B)$
from	R, S
where	$p_R \wedge p_S \wedge p_{R,S}$
group by	G
having	$H_0\left(\vec{F_0}(B)\right) \wedge H_R\left(\vec{F_R}(B)\right)$

where $\vec{F_0}$ and $\vec{F_R}$ are vectors of aggregate functions on the aggregated columns B.

An alternative way to express such a query is

select[**all** | **distinct**]	G, FB
from	R', S
where	$c_S \wedge c_{R,S} \wedge H_0(F_0B)$
where	$R'(\alpha_R, FB, F_0B) \equiv$
select all	$\alpha_R, \vec{F}(B)$ as $FB, \vec{F_0}(B)$ as F_0B
from	R
where	c_R
group by	α_R
having	$H_R\left(\vec{F_R}(B)\right)$

The according equivalence is [912]:

$$\Pi_{G,F}\left(\sigma_{H_R\wedge H_0}\left(\Gamma_{G;F:\vec{F}(B),F_R:\vec{F_R}(B),F_0:\vec{F_0}(B)}\left(\sigma_{p_R\wedge p_S\wedge p_{R,S}}(R\times S)\right)\right)\right)$$
$$\equiv$$
$$\Pi_{G,F}\left(\sigma_{p_{R,S}\wedge p_S\wedge H_0(F_0)}\right)\left(\Pi_{G,F,F_0}\left(\sigma_{H_R}\left(\Gamma_{G;F:\vec{F}(B),F_R:\vec{F_R}(B)F_0:\vec{F_0}(B)}(R)\right)\right)\times S\right)$$

20.3 Coalescing Grouping

In this section we introduce *coalescing grouping* which slightly generalizes *simple coalescing grouping* as introduced in [142].

We first illustrate the main idea by means of an example.
Given two relation schemes

Sales (pid, deptid, total_price)
Department (did, name, region)

the query

select	region, **sum** (total_price) as s
from	Sales, Department
where	did = deptid
group by	region

is straightforwardly translated into the following algebraic expression:

$$\Gamma_{region;s:sum(total_price)}(\text{Sales} \bowtie_{deptid=did} \text{Department})$$

Note that Equivalence **??** cannot be applied here. However, if there are many sales performed by a department, it might be worth reducing the cardinality of the left join input by introducing an additional group operator. The result is

$$\Gamma_{region;s=sum(s')}\left(\Gamma_{deptid;s':sum(total_price)}(\text{Sales}) \bowtie_{deptid=did} \text{Department}\right)$$

Note that we must keep the outer grouping.

That is, we introduced an additional group operator to reduce the cardinality of sales. This way, all subsequent joins (only one in this case) become cheaper and the additional group operator may result in a better plan.

We have the following restrictions for this section:

1. There are no NULL-values allowed for attributes occurring in the query.
2. All queries are of the form **select all**.
   That is **select distinct** is not allowed.
3. All aggregate functions agg must fulfill $\text{agg}s_1 \cup s_2 = \text{agg}\{\text{agg}(s_1), \text{agg}(s_2)\}$ for bags s_1 and s_2.
   This has two consequences:
   - Allowed are only sum, min, max. Not allowed are avg and count.
   - For any allowed aggregate function we only allow for **agg(all** ...**)**. Forbidden is **agg(distinct** ...**)**.
4. The query is a single-block conjunctive query with no **having** and no **order by** clause.

The above transformation is an application of the following equivalence, where R_1 and R_2 can be arbitrary algebraic expressions:

$$\Gamma_{G;A}\left(R_1 \bowtie_p R_2\right) \equiv \Gamma_{G;A_2}\left(\Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2\right) \tag{20.1}$$

with

$$\begin{aligned} A &= A_1 : agg_1\left(e_1\right), \ldots, A_n : agg_n\left(e_n\right) \\ A_1 &= A_1^1 : agg_1^1\left(e_1\right), \ldots, A_n^1 : agg_n^1\left(e_n\right) \\ A_2 &= A_1 : agg_1^2\left(A_1^1\right), \ldots, A_n : agg_n^2\left(A_n^2\right) \end{aligned}$$

$$G_1 = \left(\mathcal{F}\left(p\right) \cup G\right) \cap \mathcal{A}\left(R_1\right)$$

Further, the following condition must hold for all $i (1 \leq i \leq n)$:

$$agg_i\left(\bigcup_k S_k\right) = agg_i^2\left(\bigcup_k \{agg_i^1\left(S_i\right)\}\right)$$

In the above example, we had $agg_1 = agg_1^1 = agg_1^2 =$ **sum**.

We now prove the correctness of Equivalence 20.1.

Proof:
First, note that

$$R_1 \bowtie_p R_2 = \bigcup_{t_2 \in R_2} R_1 \bowtie_p \{t_2\} \tag{20.2}$$

Second, note that for a given t_2

$$\begin{aligned} \Gamma_{G;A}\left(R_1[t_1]\right) \bowtie_p \{t_2\} &= \sigma_{p(t_1 \circ t_2)}\left(\Gamma_{G;A}\left(R_1\left[t_1\right]\right)\right) \\ &= \Gamma_{G;A}\left(\sigma_{p(t_1 \circ t_2)}\left(R_1\left[t_1\right]\right)\right) \\ &= \Gamma_{G;A}\left(R_1\left[t_1\right] \bowtie_p \{t_2\}\right) \end{aligned} \tag{20.3}$$

holds where we have been a little sloppy with t_1. Applying (20.2) and (20.3) to $\Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2$, the inner part of the right-hand side of the equivalence yields:

$$\begin{aligned} \Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p R_2 &= \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}\left(R_1\right) \bowtie_p \{t_2\} \\ &= \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}\left(R_1 \bowtie_p \{t_2\}\right) \end{aligned} \tag{20.4}$$

Call the last expression X.

Then the right-hand side of our equivalence becomes

$$\begin{aligned} \Gamma_{G;A_2}(X) = \{ & t \circ a_2 | t \in \Pi_G(X), a_2 = \left(A_1 : a_1^2, \ldots, A_n : a_n^2\right), \\ & a_i^2 = agg_i^2\left(\left\{s.A_i^1 | s \in X, S|_G = t\right\}\right)\} \end{aligned} \tag{20.5}$$

Applying (20.2) to the left-hand side of the equivalence yields:

$$\Gamma_{G;A}\left(R_1 \bowtie_p R_2\right) = \Gamma_{G;A}\left(\bigcup_{t_2 \in R_2} \left\{t_1 \circ t_2 | t_1 \in R_1, p\left(t_1 \circ t_2\right)\right\}\right) \tag{20.6}$$

Abbreviate $\bigcup_{t_2 \in R_2} \left\{t_1 \circ t_2 | t_1 \in R_1, p\left(t_1 \circ t_2\right)\right\}$ by Y.
Applying the definition of $\Gamma_{G;A}$ yields:

$$\begin{aligned} \{t \circ a \mid & t \in \Pi_G(Y), a = \left(A_1 : e_1, \ldots, A_n : e_n\right), \\ & a_i = agg_i\left(\left\{e_i(s) | s \in Y, S|_G = t\right\}\right)\} \end{aligned} \tag{20.7}$$

Compare (20.5) and (20.7). Since $\Pi_G(X) = \Pi_G(Y)$, they can only differ in their values of A_i.

Hence, it suffices to prove that $a_i^2 = a_i$ for $1 \leq i \leq n$ for any given t.

$$
\begin{aligned}
a_i^2 &= agg_i^2(\{s.A_i^1 | s \in X, S|_G = t\}) \\
&= agg_i^2(\{s.A_i^1 | s \in \bigcup_{t_2 \in R_2} \Gamma_{G_1;A_1}(R_1 \ltimes_p \{t_2\}), S|_G = t\}) \\
&= agg_i^2(\{s.A_i^1 | s \in \bigcup_{t_2 \in R_2} \{t_1 \circ t_2 \circ a_1 | t_1 \in \Pi_{G_1}(R_1), p(t_1 \circ t_2), \\
&\qquad\qquad a_1 = (A_1^1 : a_1^1, \ldots, A_n^1 : a_n^1) \\
&\qquad a_i^1 = agg_i^1(\{e_i(s_1 \circ t_2) | s_1 \in R_1, S_1|_{G_1 = t_1}, p(s_1, t_2)\}) \\
&\qquad S|_G = t\}\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(s_1 \circ t_2) | t_1 \in \Pi_{G_1}(R_1), p(t_1 \circ t_2), s_1 \in R_1, S_1|_{G_1} = t_1, \\
&\qquad\qquad p(s_1, t_1), t_1 \circ t_2|_G = t\})\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(s_1 \circ t_2) | s_1 \in R_1, s_1 \circ t_2|_G = t, p(s_1 \circ t_2)\})\}) \\
&= agg_i^2(\bigcup_{t_2 \in R_2} \{agg_i^1(\{e_i(t_1 \circ t_2) | t_1 \in R_1, t_1 \circ t_2|_G = t, p(t_1 \circ t_2)\})\}) \\
&= agg_i(\bigcup_{t_2 \in R_2} \{e_i(t_1 \circ t_2) | t_1 \in R_1, p(t_1 \circ t_2), t_1 \circ t_2|_G = t\}) \\
&= agg_i(\{e_i(s) | s \in \bigcup_{t_2 \in R_2} \{t_1 \circ t_2 | t_1 \in R_1, p(t_1 \circ t_2)\}, S|_G = t\}) \\
&= a_i
\end{aligned}
$$

Equivalence 20.1 can be used to add an additional coalescing grouping in front of any of a sequence of joins. Consider the schematic operator tree in Figure 20.2(a). It is equivalent to the one in (b), which in turn is equivalent to the one in (c) if the preconditions of Equivalence 20.1 hold. Performing a similar operation multiple times, any of the join operations can be made to be preceded by a coalescing grouping.

20.4 ToDo

[688]

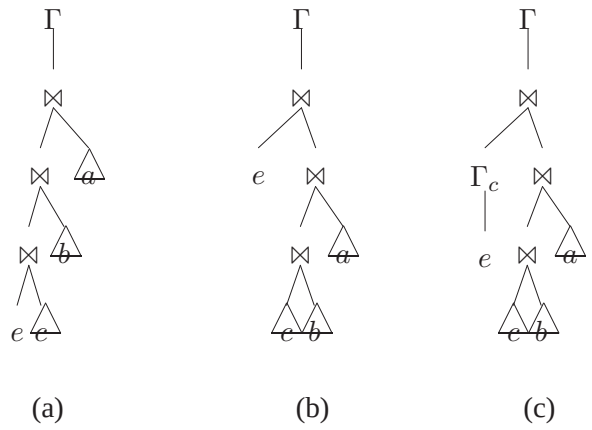


Figure 20.2: Applications of coalescing grouping

Part V

Plan Generation

Chapter 21

Introduction to Plan Generation

21.1 Search Space Selection

so far: see basics. now: more operators, more possibilities beyond freely reorderability. select? customize?

21.2 Complexity Results

All SQL queries can be processed in polynomial time in database size:[129]. a large subclass can be processed in quasi-linear time:[891]

is the result empty is PSPACE complete:[128]. for conjunctive queries: NP-complete for Blockwise nested n-ary loop joins [428]. Complexity of transformation-based join enumeration [656]. See also Chapter 3

21.3 Implementation Approaches and Architectures

21.3.1 Hard-wired Algorithms

21.3.2 Rule Based Approaches

Earliest Approach: [794] rule sets: [700, 163]

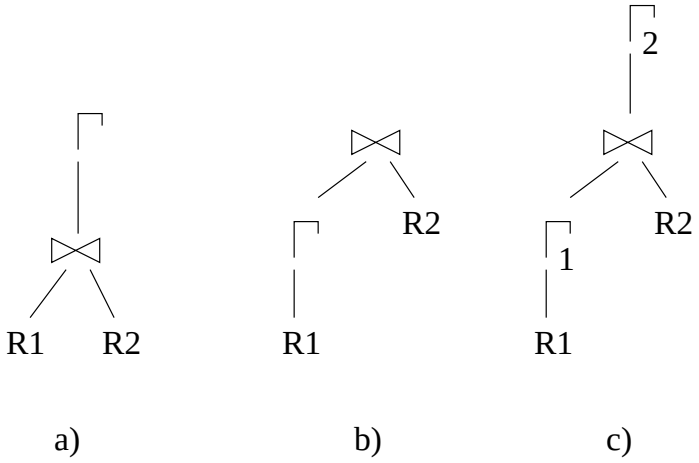


Figure 21.1: Early grouping

21.3.3 Blackboard Architecture

21.4 Index Selection

21.5 Disjunctive Queries

21.6 Outer Joins

21.7 Plan Improvements, Postprocessing, and Polishing

21.7.1 Pushing group operators

Traditionally, group operators are performed last. Lately, it was pointed out that under certain conditions, a group operator can be pushed down a query plan resulting in cheaper plans [142, 143, 910, 911, 913, 912]. The saving is due to the fact that a grouping reduces cardinality and hence the costs of subsequent joins decreases.

Consider Fig. 21.1. Plan a) is a regular plan where the group operator is the top-level operator. Under the following conditions plan b) is equivalent to plan a):

- The join is an equi join on the foreign key attributes of R_1 and the key attribute in R_2.
- The aggregate functions only involve attributes of R_1.
- The set of grouping attributes is a subset of the attributes of R_1.
- The set of grouping attributes is a not necessarily strict superset of the foreign key attributes of R_1.

A typical example query fulfilling these conditions is

select d.name, sum(e.salary)
from Employee e, Department d

where e.dno = d.dno
group bye.dno

Two remarks are in order. First, note that in case the grouping is on *d.dno*, the conditions are violated but nevertheless the grouping can be performed. In order to discover this, the equivalence relation on attributes (as defined by the transitivity of "=" conditions in the **where** clause) must be used. Second, note that this is not an SQL query since we project on attributes that are not mentioned in the **group by** clause. However, the query compiler can easily establish that *d.name* is functionally dependend on *d.dno* (the primary key of *Department*) which is in the same equivalence class a *e.dno*. The correctly formulated SQL query can also be rewritten by a similar line of argument. This example illustrates, that rewriting often necessitates a lot of reasoning to be performed by the query compiler.

The plan c) illustrates the case where the top-level grouping cannot be eliminated, but still an early grouping can be performed to reduce the size of one of the join partners. For this to be possible, the aggregate functions involved must be *decomposable*, that is the aggregate *agg*$(S_1 \cup S_2)$ must be computable from aggregates *agg'*(S_1) and *agg'*(S_2).

Of course, separating join ordering from pushing down grouping operators might miss the best plan. However, there is no known algorithm for plan generation that consideres early grouping while constructing plans. The techniques for pushing down grouping also apply to **select distinct** queries since they are a special case of grouping.

More details can be found in Section 6.2.12.

21.7.2 Predicate pull-up

If the plan generator does not consider selections, that is selections are pushed down, then there might be a need to pull up expensive predicates at this stage. Expensive selections and join operations are reordered in search for a cheaper plan. Again, considering expensive predicates as late as in Rewrite II might loose optimal plans.

21.7.3 Polishing

The task of polishing is to clash sequences of operators together. Equivalences like $\sigma_p(\sigma_q(e)) = \sigma_{p \wedge q}(e)$ are applied in order to reduce the number of algebraic operators. Further, projections (that do not eliminate duplicates) are pushed down as far as possible.

21.7.4 Optimizing complex boolean expressions

If there is a selection predicate that is a complex predicate consisting of disjunctions and/or conjunctions of base predicates of different costs and selectivities, then there might be a need to optimize the complex selection predicate. A heuristic for doing so is presented in [479]. see also Chapter **??**

21.8 Bibliography

[692, 693, 703, 706]

Chapter 22

Hard-Wired Algorithms

22.1 Hard-wired Dynamic Programming

22.1.1 Introduction

Plan generation is performed block-wise. The goal is to generate a plan for every block. Typically, not all possible plans are generated. For example, the group operator (if necessary for the query) is mostly performed last (see also Sec. **??**). This mainly leaves ordering joins and selections as the task of plan generation. A plan is an operator tree whose node consist of physical algebraic operators, e.g. selection, sort-operator, sort-merge and other joins, relation and index scans. The process of plan generation has received a lot of attention. Often, the term query optimization is used synonymous for the plan generation phase.

Figure 22.1 shows a plan for the block

select e.name
from Employee e, Department d
where e.dno = d.dno and d.name = “shoe”

The bottom level contains two table scans that scan the base tables *Employee* and *Department*. Then, a selection operator is applied to restrict the departments to those named *“shoe”*. A nested-loop join is used to select those employees that work in the selected departments. The projection restricts the output to the name of the employees, as required by the query block. For such a plan, a *cost function* is used to estimate its cost. The goal of plan generation is to generate the cheapest possible plan. Costing is briefly sketched in Section **??**.

The foundation of plan generation are algebraic equivalences. For $e, e_1, e_2, \ldots$

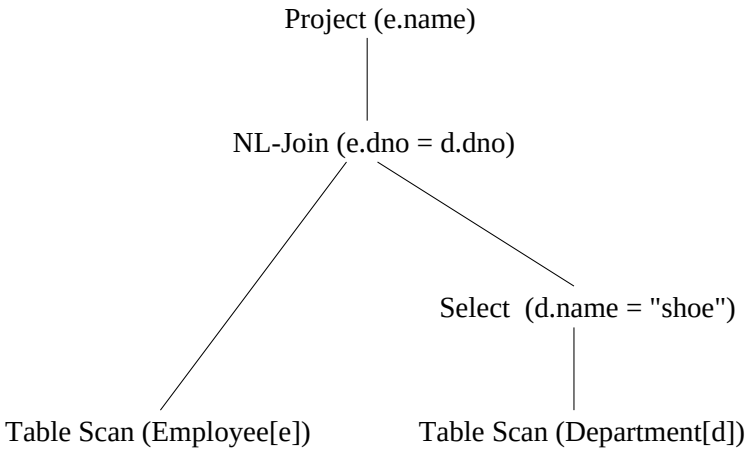


Figure 22.1: A sample execution plan

being algebraic expressions and p, q predicates, here are some example equivalences:

$$
\begin{aligned}
\sigma_p(\sigma_q(e)) &\equiv \sigma_q(\sigma_p(e)) \\
\sigma_p(e_1 \bowtie_q e_2) &\equiv (\sigma_p(e_1)) \bowtie_q e_2 \quad \text{if } p \text{ is applicable to } e_1 \\
e_1 \bowtie_p e_2 &\equiv e_2 \bowtie_p e_1 \\
(e_1 \bowtie_p e_2) \bowtie_q e_3 &\equiv e_1 \bowtie_p (e_2 \bowtie_q e_3) \\
e_1 \cup e_2 &\equiv e_2 \cup e_1 \\
(e_1 \cup e_2) \cup e3 &\equiv e_1 \cup (e_2 \cup e_3) \\
e_1 \cap e_2 &\equiv e_2 \cap e_1 \\
(e_1 \cap e_2) \cap e3 &\equiv e_1 \cap (e_2 \cap e_3) \\
\sigma_p(e_1 \cap e_2) &\equiv \sigma_p(e_1) \cap e_2
\end{aligned}
$$

For more equivalences and conditions that ought to be attached to the equivalences see the appendix 6.2. Note that commutativity and associativity of the join operator allow an arbitrary ordering. Since the join operator is the most expensive operation, ordering joins is the most prominent problem in plan generation.

These equivalences are of course independent of the actual implementation of the algebraic operators. The total number of plans equivalent to the original query block is called the *potential search space*. However, not always is the total search space considered. The set of plans equivalent to the original query considered by the plan generator is the *actual search space*. Since the System R plan generator [750], certain restrictions are applied. The most prominent are:

- Generate only plans where selections are pushed down as far as possible.
- Do not consider cross products if not absolutely necessary.
- Generate only left-deep trees.
- If the query block contains a grouping operation, the group operator is performed last.

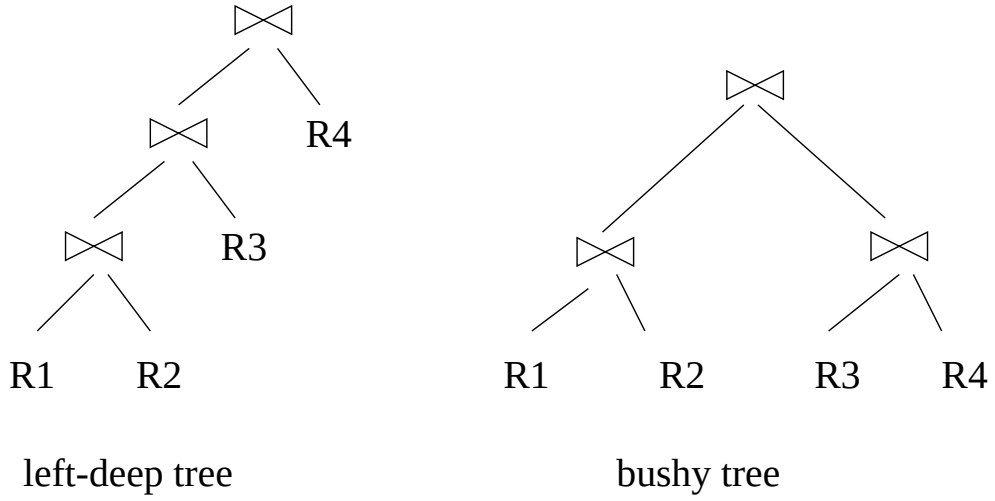


Figure 22.2: Different join operator trees

Some comments are in order. Cross products are only necessary, if the query graph is unconnected where a query graph is defined as follows: the nodes are the relations and the edges correspond to the predicates (*boolean factors*[1]) found in the **where** clause. More precisely, the query graph is a hypergraph, since a boolean factor may involve more than two relations. A left-deep tree is an operator tree where the right argument of a join operator always is a base relation. A plan with join operators whose both arguments are derived by other join operators is called *bushy tree*. Figure 22.2 gives an example of a left-deep tree and a bushy tree.

If we take all the above restrictions together, the problem boils down to ordering the join operators or relations. This problem has been studied extensively. The complexity of finding the best (according to some cost function) ordering (operator tree) was first studied by Ibaraki and Kameda [428]. They proved that the problem of generating optimal left-deep trees with no cross products is NP-hard for a special block-wise nested loop join cost function. This cost function applied in the proof is quite complex. Later is was shown that even if the cost function is very simple, the problem remains NP-hard [190]. The cost function (C_{out}) used there just adds up intermediate results sizes. This cost function is interesting in that it is the kernel of many other cost functions and it fulfills the ASI property of which we now the following: If the cost function fulfills the *ASI property* and the query graph is acyclic, then the problem can be solved in polynomial time [428, 503]. Ono and Lohman gave examples that considering cross products can substantially improve performance [627]. However, generating optimal left-deep trees with cross products even for C_{out} makes the problem NP-hard [190]. Generating optimal bushy trees is even harder. Even if there is no predicate, that is only cross products have to be used, the problem is NP-hard [735]. This is surprising since generating left-deep trees with cross products as the only operation is very simple: just sort the relations by increasing sizes.

Given the complexity of the problem, there are only two alternatives to generate plans: either explore the total search space or use heuristics. The former can be quite

[1] A *boolean factor* is a disjunction of basic predicates in a conjunctive normal form.

expensive. This is the reason why the above mentioned restrictions to the search space have traditionally been applied. The latter approach risks missing good plans. The best-known heuristics is to join the relation next, that results in the smallest next intermediate result. Estimating the cardinality of such results is discussed in Section **??**.

Traditionally, selections where pushed as far down as possible. However, for expensive selection predicates (e.g. user defined predicates, those involving user-defined functions, predicates with subqueries) this does not suffice. For example, if a computer vision application has to compute the percentage of snow coverage for a given set of satellite images, this is not going to be cheap. In fact, it can be more expensive than a join operation. In these cases, pushing the expensive selection down misses good plans. That is why lately research started to take expensive predicates into account. However, some of the proposed solutions do not guarantee to find the optimal plans. Some approaches and their bugs are discussed in [145, 406, 404, 734, 736]. Although we will subsequently give an algorithm that incorporates correct predicate placement, not all plan generators do so. An alternative approach (though less good) is to pull-up expensive predicates in the Rewrite-II-phase.

There are several approaches to explore the search space. The original approach is to use dynamic programming [750]. The dynamic programming algorithm is typically hard-coded. Figure 22.3 illustrates the principle of bottom-up plan generation as applied in dynamic programming. The bottom level consists of the original relations to be joined. The next level consists of all plans that join a subset of cardiniality two of the original relations. The next level contains all plans for subsets of cardinality three, and so on. With the advent of new query optimization techniques, new data models, extensible database systems, researchers where no longer satisfied with the hard-wired approach. Instead, they aimed for rule-based plan generation. There exist two different approaches for rule-based query optimizers. In the first approach, the algebraic equivalences that span the search space are used to transform some initial query plan derived from the query block into alternatives. As search strategies either exhaustive search is used or some stochastic approach such as simulated annealing, iterative improvement, genetic algorrithms and the like [70, 433, 438, 439, 803, 829, 828, 831]. This is the *transformation-based* approach. This approach is quite inefficient. Another approach is to generate plans by rules in a bottom-up fashion. This is the *generation-based* approach. In this approach, either a dynamic programming algorithm [552] is used or memoization [350]. It is convenient to classify the rules used into logical and physical rules. The logical rules directly reflect the algebraic equivalences. The physical rules or implementation rules transform a logical algebraic operator into a physical algebraic operator. For example, a join-node becomes a nested-loop join node.

22.1.2 A plan generator for bushy trees

Within the brief discussion in the last subsection, we enumerated plans such that first all 1-relation plans are generated, then all 2-relation plans and so on. This enumeration order is not the most efficient one. Let us consider the simple problem where we have to generate exactly one best plan for the subsets of the n element set of relations to be joined. The empty subset is not meaningful, leaving the number of subsets to be investigated at $2^n - 1$. Enumerating these subsets can be done most efficient by enumerating them in *counting order*. That is, we initialize a n bit counter with 1 and

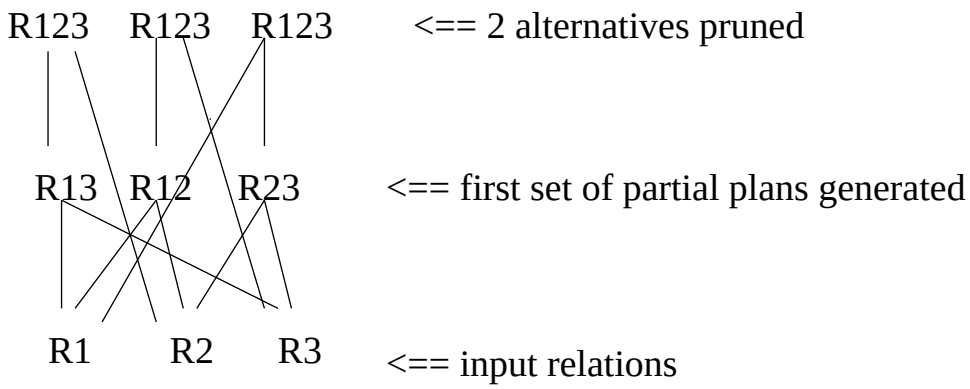


Figure 22.3: Bottom up plan generation

count until have reached $2^n - 1$. The n bits represent the subsets. Note that with this enumeration order, plans are still generated bottom up. For a given subset R of the relations (encoded as the bit pattern a), we have to generate a plan from subsets of this subset (encoded as the bit pattern s). For example, if we only want to generate left-deep trees, then we must consider 1 element subsets and their complements. If we want to generate bushy trees, all subsets must be considered. We can generate these subsets by a very fast algorithm developed by Vance and Maier [860]:

```
s = a & -a;
while(s) {
   s = a & (s - a);
   process(s);
}
```

The meaning of *process(s)* depends on the kinds of plans we generate. If we concentrate on join ordering neglecting selection operations (i.e. pushing them) This step essentially looks up the plans for s and its complement $\overline{s}$ and then joins the plans found there. Lookup is best implemented via an array with s as an index.

22.1.3 A plan generator for bushy trees and expensive selections

Figure 22.4 shows the pseudocode of a dynamic programming algorithm that generates plans with cross products, selections, and joins. It generates optimal bushy trees. Efficient implementation technique for the algorithm can be found in [860, 736]. As input parameters, the algorithm takes a set of relations R and a set of predicates P. The set of relations for which a selection predicate exists is denoted by R_S. We identify relations and predicates that apply to these relations. For all subsets M_k of the relations and subsets P_l of the predicates, an optimal plan is constructed and entered into the table T. The loops range over all M_k and P_l. Thereby, the set M_k is split into two disjoint subsets L and L', and the set P_l is split into three parts (line 7). The first part (V) contains those predicates that apply to relations in L only. The second part (V') contains those predicates that apply to relations in L' only. The third part (p) is a conjunction of all the join predicates connecting relations in L and L' (line 8). Line 9 constructs a plan by joining the two plans found for the pairs $[L, V]$ and $[L', V']$ in

```
proc Optimal-Bushy-Tree(R, P)
1  for k = 1 to n do
2      for all k-subsets M_k of R do
3          for l = 0 to min(k, m) do
4              for all l-subsets P_l of M_k ∩ R_S do
5                  best_cost_so_far = ∞;
6                  for all subsets L of M_k with 0 < |L| < k do
7                      L' = M_k \ L, V = P_l ∩ L, V' = P_l ∩ L';
8                      p = ⋀{p_{i,j} | p_{i,j} ∈ P, R_i ∈ V, R_j ∈ V'};   // p=true might hold
9                      T = (T[L, V] ⋈_p T[L', V']);
10                     if Cost(T) < best_cost_so_far then
11                         best_cost_so_far = Cost(T);
12                         T[M_k, P_l] = T;
13                     fi;
14                 od;
15                 for all R ∈ P_l do
16                     T = σ_R(T[M_k, P_l \ {R}]);
17                     if Cost(T) < best_cost_so_far then
18                         best_cost_so_far = Cost(T);
19                         T[M_k, P_l] = T;
20                     fi;
21                 od;
22             od;
23         od;
24     od;
25 od;
26 return T[R, S];
```

Figure 22.4: A Dynamic Programming Optimization Algorithm

the table T. If this plan has so far the best costs, it is memoized in the table (lines 10-12). Last, different possibilities of not pushing predicates in P_l are investigated (lines 15-19).

Another issue that complicates the application of dynamic programming are certain properties of plans. The most prominent such properties are *interesting orders* [750, 788, 789]. Take a look at the following query:

```
select   d.no, e.name
from     Employee e, Department d
where    e.dno = d.dno
order by d.dno
```

Here, the user requests the result to be order on *d.dno*. Incidentally, this is also a join attribute. During bottom up plan generation, we might think that a Grace hash join is more efficient than a sort-merge join since the cost of sorting the relations is too high. However, the result has to be sorted anyway so that this sort may pay off. Hence, we have have to keep both plans. The approach is the following. In the example, an ordering on *d.dno* is called an interesting order. In general, any order that is helpful for

ordering the output as requested by the user, for a join operator, for a grouping operator, or for duplicate elimination is called an *interesting order*. The dymamic programming algorithm is then modified such that plans are not pruned, if they produce different interesting orders.

22.1.4 A plan generator for bushy trees, expensive selections and functions

22.2 Bibliography

Chapter 23

Rule-Based Algorithms

23.1 Rule-based Dynamic Programming

The section is beyond the scope of the paper and the reader is refered to the starburst papers, especially [379, 524, 523, 552, 553].

23.2 Rule-based Memoization

This section is beyond the scope of the paper and the reader is refered to the Volcano and Cascade papers [336, 341, 346, 349, 350]. Both optimizer frameworks derived from the earlier Exodus query optimizer generator [334, 347].

23.3 Bibliography

Chapter 24

Deriving and Dealing with Interesting Orderings and Groupings

[This chapter was written by Thomas Neumann and Guido Moerkotte]

24.1 Introduction

The most expensive operations (e.g. join, grouping, duplicate elimination) during query evaluation can be performed more efficiently if the input is ordered or grouped in a certain way. Therefore, it is crucial for query optimization to recognize cases where the input of an operator satisfies the ordering or grouping requirements needed for a more efficient evaluation. Since a plan generator typically considers millions of different plans – and, hence, operators –, this recognition easily becomes a performance bottleneck for plan generation, often leading to heuristic solutions.

The importance of exploiting available orderings has already been recognized in the seminal work of Selinger et al [750]. They presented the concept of interesting orderings and showed how redundant sort operations could be avoided by reusing available orderings, rendering sort-based operators like sort-merge join much more interesting.

Along these lines, it is beneficial to reuse available grouping properties, for example for hash-based operators. While heuristic techniques to avoid redundant group-by operators have been given [142], for a long time groupings have not been treated as thoroughly as orderings. One reason might be that while orderings and groupings are related (every ordering is also a grouping), groupings behave somewhat differently. For example, a tuple stream grouped on the attributes $\{a, b\}$ need not be grouped on the attribute $\{a\}$. This is different from orderings, where a tuple stream ordered on the attributes (a, b) is also ordered on the attribute (a). Since no simple prefix (or subset) test exists for groupings, optimizing groupings even in a heuristic way is much more difficult than optimizing orderings. Still, it is desirable to combine order optimization and the optimization of groupings, as the problems are related and treated similarly during plan generation. Recently, some work in this direction has been published

[876]. However, this only covers a special case of grouping. Instead, in this chapter we follow the approach presented by Neumann and Moerkotte [618, 617]

Other existing frameworks usually consider only order optimization, and experimental results have shown that the costs for order optimization can have a large impact on the total costs of query optimization [618]. Therefore, some care is needed when adding groupings to order optimization, as a slowdown of plan generation would be unacceptable.

In this chapter, we present a framework to efficiently reason about orderings and groupings. It can be used for the plan generator described in Chapter **??**, but is actually an independent component that could be used in any kind of plan generator. Experimental results show that it efficiently handles orderings and groupings at the same time, with no additional costs during plan generation and only modest one time costs. Actually, the operations needed for both ordering and grouping optimization during plan generation can be performed in $O(1)$, basically allowing to exploit groupings for free.

24.2 Problem Definition

The order manager component used by the plan generator combines order optimization and the handling of grouping in one consistent set of algorithms and data structures. In this section, we give a more formal definition of the problem and the scope of the framework. First, we define the operations of ordering and grouping (Section 24.2.1 and 24.2.2). Then, we briefly discuss functional dependencies (Section 24.2.3) and how they interact with algebraic operators (Section 24.2.4). Finally, we explain how the component is actually used during plan generation (Section 24.2.5).

24.2.1 Ordering

During plan generation, many operators require or produce certain orderings. To avoid redundant sorting, it is required to keep track of the orderings a certain plan satisfies. The orderings that are relevant for query optimization are called *interesting orders* [750]. The set of *interesting orders* for a given query consists of

1. all orderings required by an operator of the physical algebra that may be used in a query execution plan for the given query, and

2. all orderings produced by an operator of the physical algebra that may be used in a query execution plan for the given query.

This includes the final ordering requested by the given query, if this is specified.

The interesting orders are *logical orderings*. This means that they specify a condition a tuple stream must meet to satisfy the given ordering. In contrast, the *physical ordering* of a tuple stream is the actual succession of tuples in the stream. Note that while a tuple stream has only one physical ordering, it can satisfy multiple logical orderings. For example, the stream of tuples $((1,1),(2,2))$ with schema (a,b) has one physical ordering (the actual stream), but satisfies the logical orderings a, b, ab and ba.

Some operators, like `sort`, actually influence the physical ordering of a tuple stream. Others, like `select`, only influence the logical ordering. For example, a

`sort[a]` produces a tuple stream satisfying the ordering (a) by actually changing the physical order of tuples. After applying `select[a=b]` to this tuple stream, the result satisfies the logical orderings $(a), (b), (a, b), (b, a)$, although the physical ordering did not change. Deduction of logical orderings can be described by using the well-known notion of *functional dependency* (FD) [788]. In general, the influence of a given algebraic operator on a set of logical orderings can be described by a set of functional dependencies.

We now formalize the problem. Let $R = (t_1, \ldots, t_r)$ be a stream (ordered sequence) of tuples in attributes $A_1, \ldots, A_n$. Then R *satisfies the logical ordering* $o = (A_{o_1}, \ldots, A_{o_m})$ ($1 \leq o_i \leq n$) if and only if for all $1 \leq i < j \leq r$ the following condition holds:

$$
\begin{array}{lll}
 & (t_i.A_{o_1} \leq t_j.A_{o_1}) & \\
\wedge \quad \forall 1 < k \leq m & & (\exists 1 \leq l < k(t_i.A_{o_l} < t_j.A_{o_l})) \vee \\
 & & ((t_i.A_{o_{k-1}} = t_j.A_{o_{k-1}}) \wedge \\
 & & (t_i.A_{o_k} \leq t_j.A_{o_k}))
\end{array}
$$

Next, we need to define the inference mechanism. Given a logical ordering $o = (A_{o_1}, \ldots, A_{o_m})$ of a tuple stream R, then R obviously satisfies any logical ordering that is a prefix of o including o itself.

Let R be a tuple stream satisfying both the logical ordering $o = (A_1, \ldots, A_n)$ and the functional dependency $f = B_1, \ldots, B_k \rightarrow B_{k+1}$[1] with $B_i \in \{A_1 \ldots A_n\}$. Then R also satisfies any logical ordering derived from o as follows: add B_{k+1} to o at any position such that all of $B_1, \ldots, B_k$ occurred before this position in o. For example, consider a tuple stream satisfying the ordering (a, b); after inducing the functional dependency $a, b \rightarrow c$, the tuple stream also satisfies the ordering (a, b, c), but not the ordering (a, c, b). Let O' be the set of all logical orderings that can be constructed this way from o and f after prefix closure. Then, we use the following notation: $o \vdash_f O'$. Let e be the equation $A_i = A_j$. Then, $o \vdash_e O'$, where O' is the prefix closure of the union of the following three sets. The first set is O_1 defined as $o \vdash_{A_i \rightarrow A_j} O_1$, the second is O_2 defined as $o \vdash_{A_j \rightarrow A_i} O_2$, and the third is the set of logical orderings derived from o where a possible occurrence of A_i is replaced by A_j or vice versa. For example, consider a tuple stream satisfying the ordering (a); after inducing the equation $a = b$, the tuple stream also satisfies the orderings $(a, b), (b)$ and (b, a). Let e be an equation of the form $A = const$. Then O' ($o \vdash_e O'$) is derived from o by inserting A at any position in o. This is equivalent to $o \vdash_{\emptyset \rightarrow A} O'$. For example, consider a tuple stream satisfying the ordering (a, b); after inducing the equation $c = const$ the tuple stream also satisfies the orderings $(c, a, b), (a, c, b)$ and (a, b, c).

Let O be a set of logical orderings and F a set of functional dependencies (and possibly equations). We define the sets of inferred logical orderings $\Omega_i(O, F)$ as fol-

[1] Any functional dependency which is not in this form can be normalized into a set of FDs of this form.

lows:

$$\begin{aligned}\Omega_0(O,F) &:= O \\ \Omega_i(O,F) &:= \Omega_{i-1}(O,F) \cup \\ &\qquad \bigcup_{f\in F, o\in\Omega_{i-1}(O,F)} O' \text{ with } o \vdash_f O'\end{aligned}$$

Let $\Omega(O,F)$ be the prefix closure of $\bigcup_{i=0}^{\infty}\Omega_i(O,F)$. We write $o \vdash_F o'$ if and only if $o' \in \Omega(O,F)$.

24.2.2 Grouping

It was shown in [876] that, similar to order optimization, it is beneficial to keep track of the groupings satisfied by a certain plan. Traditionally, group-by operators are either applied after the rest of the query has been processed or are scheduled using some heuristics [142]. However, the plan generator could take advantage of grouping properties produced e.g. by avoiding re-hashing if such information was easily available.

Analogous to order optimization, we call this *grouping optimization* and define that the set of *interesting groupings* for a given query consists of

1. all groupings required by an operator of the physical algebra that may be used in a query execution plan for the given query
2. all groupings produced by an operator of the physical algebra that may be used in a query execution plan for the given query.

This includes the grouping specified by the group-by clause of the query, if any exists.

These groupings are similar to logical orderings, as they specify a condition a tuple stream must meet to satisfy a given grouping. Likewise, functional dependencies can be used to infer new groupings.

More formally, a tuple stream $R = (t_1, \ldots, t_r)$ in attributes $A_1, \ldots, A_n$ *satisfies the grouping* $g = \{A_{g_1} \ldots, A_{g_m}\}$ $(1 \leq g_i \leq n)$ if and only if for all $1 \leq i < j < k \leq r$ the following condition holds:

$$\begin{aligned}& \forall 1 \leq l \leq m \quad t_i.A_{g_l} = t_k.A_{g_l} \\ \Rightarrow\ & \forall 1 \leq l \leq m \quad t_i.A_{g_l} = t_j.A_{g_l}\end{aligned}$$

Two remarks are in order here. First, note that a grouping is a set of attributes and not – as orderings – a sequence of attributes. Second, note that given two groupings g and $g' \subset g$ and a tuple stream R satisfying the grouping g, R need not satisfy the grouping g'. For example, the tuple stream $((1,2),(2,3),(1,4))$ with the schema (a,b) is grouped by $\{a,b\}$, but not by $\{a\}$. This is different from orderings, where a tuple stream satisfying an ordering o also satisfies all orderings that are a prefix of o.

New groupings can be inferred by functional dependencies as follows: Let R be a tuple stream satisfying both the grouping $g = \{A_1, \ldots, A_n\}$ and the functional dependency $f = B_1, \ldots, B_k \rightarrow B_{k+1}$ with $\{B_1, \ldots, B_k\} \subseteq \{A_1, \ldots, A_n\}$. Then R also satisfies the grouping $g' = \{A_1, \ldots, A_n\} \cup \{B_{k+1}\}$. Let G' be the set of all groupings that can be constructed this way from g and f. Then we use the following

notation: $g \vdash_f G'$. For example $\{a, b\} \vdash_{a,b \to c} \{a, b, c\}$. Let e be the equation $A_i = A_j$. Then $g \vdash_e G'$ where G' is the union of the following three sets. The first set is G_1 defined as $g \vdash_{A_i \to A_j} G_1$, the second is G_2 defined as $g \vdash_{A_j \to A_i} G_2$, and the third is the set of groupings derived from g where a possible occurrence of A_i is replaced by A_j or vice versa. For example, $\{a, b\} \vdash_{b=c} \{a, c\}$. Let e be an equation of the form $A = const$. Then $g \vdash_e G'$ is defined as $g \vdash_{\emptyset \to A} G'$. For example, $\{a, b\} \vdash_{c=const} \{a, b, c\}$.

Let G be a set of groupings and F be a set of functional dependencies (and possibly equations). We define the set of inferred groupings $\Omega_i(G, F)$ as follows:

$$\begin{aligned} \Omega_0(G, F) &:= G \\ \Omega_i(G, F) &:= \Omega_{i-1}(G, F) \cup \\ &\qquad \bigcup_{f \in F, g \in \Omega_{i-1}(G,F)} G' \text{ with } g \vdash_f G' \end{aligned}$$

Let $\Omega(G, F)$ be $\bigcup_{i=0}^{\infty} \Omega_i(G, F)$. We write $g \vdash_F g'$ if and only if $g' \in \Omega(G, F)$.

24.2.3 Functional Dependencies

The reasoning about orderings and groupings assumes that the set of functional dependencies is known. The process of gathering the relevant functional dependencies is described in detail in [788, 789]. Predominantly, there are three sources of functional dependencies:

1. key constraints
2. join predicates [references constraints]
3. filter predicates
4. simple expressions

However, the algorithm makes no assumption about the functional dependencies. If for some reason an operator induces another kind of functional dependency (e.g., when using TID-based optimizations [575]), this can be handled the same way. The only important fact is that we provide the set of functional dependencies as input to the algorithm.

24.2.4 Algebraic Operators

To illustrate the propagation of orderings and groupings during query optimization, we give some rules for concrete (physical) operators in Figure 24.1. As a shorthand, we use the following notation:

$O(R)$	set of logical orderings and groupings satisfied by the physical ordering of the relation R
$O(S)$	inferred set of logical orderings and groupings satisfied by the tuple stream S
$x \downarrow$	$\{y \mid y \in x\}$

operator	requires	produces
scan(R)	-	$O(R)$
indexscan(Idx)	-	$O(Idx)$
map(S,$a = f(b)$)	-	$\Omega(O(S), b \rightarrow a)$
select(S,$a = b$)	-	$\Omega(O(S), a = b)$
bnl-join(S_1,S_2)	-	$O(S_1)$
indexnl-join(S_1,S_2)	-	$O(S_1)$
djoin(S_1,S_2)	-	$O(S_1)$
sort(S,$a_1, \ldots, a_n$)	-	$(a_1, \ldots, a_n)$
group-by(S,$a_1, \ldots, a_n$)	-	$\{a_1, \ldots, a_n\}$
hash(S,$a_1, \ldots, a_n$)	-	$\{a_1, \ldots, a_n\}$
sort-merge(S_1,S_2,$\vec{a} = \vec{b}$)	$\vec{a} \in O(S_1) \wedge \vec{b} \in O(S_2)$	$\Omega(O(S_1), \vec{a} = \vec{b})$
hash-join(S_1,S_2,$\vec{a} = \vec{b}$)	$\vec{a} \downarrow\in O(S_1) \wedge \vec{b} \downarrow\in O(S_2)$	$\Omega(O(S_1), \vec{a} = \vec{b})$

Figure 24.1: Propagation of orderings and groupings

Note that these rules somewhat depend on the actual implementation of the operators, e.g. a blockwise nested loop join might actually destroy the ordering if the blocks are stored in hash tables. The rules are also simplified: For example, a group-by will probably compute some aggregate functions, inducing new functional dependencies. Furthermore, additional information can be derived from schema information: If the right-hand side of a dependent join (index nested loop joins are similar) produces at most one tuple, and the left-hand side is grouped on the free attributes of the right-hand side (e.g. if they do not contain duplicates) the output is also grouped on the attributes of the right-hand side. This situation is common, especially for index nested loop joins, and is detected automatically if the corresponding functional dependencies are considered. Therefore, it is important that all operators consider all functional dependencies they induce.

EXC

24.2.5 Plan Generation

To exploit available logical orderings and groupings, the plan generator needs access to the combined order optimization and grouping component, which we describe as an *abstract data type* (ADT). An instance of this abstract data type `OrderingGrouping` represents a set of logical orderings and groupings, and wherever necessary, an instance is embedded into a plan note. The main operations the abstract data type `OrderingGrouping` must provide are

1. a constructor for a given logical ordering or grouping,
2. a membership test (called `containsOrdering(LogicalOrdering)`) which tests whether the set contains the logical ordering given as parameter,
3. a membership test (called `containsGrouping(Grouping)`) which tests whether the set contains the grouping given as parameter, and
4. an inference operation (called `infer(set<FD>)`). Given a set of functional dependencies and equations, it computes a new set of logical orderings and

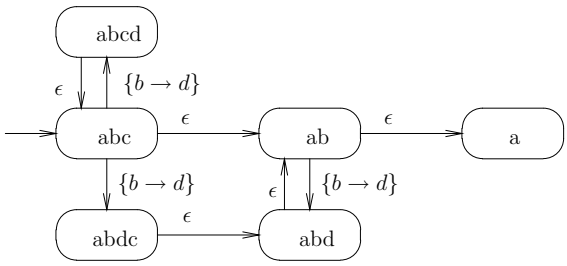


Figure 24.2: Possible FSM for orderings

groupings a tuple stream satisfies.

These operations can be implemented by using the formalism described before: `containsOrdering` tests for $o \in O$, `containsGrouping` tests for $o \in G$ and `infer(F)` calculates $\Omega(O, F)$ respectively $\Omega(G, F)$. Note that the intuitive approach to explicitly maintain the set of all logical orderings and groupings is not useful in practice. For example, if a sort operator sorts a tuple stream on (a, b), the result is compatible with logical orderings $\{(a, b), (a)\}$. After a selection operator with selection predicate $x = const$ is applied, the set of logical orderings changes to $\{(x, a, b), (a, x, b), (a, b, x), (x, a), (a, x), (x)\}$. Since the size of the set increases quadratically with every additional selection predicate of the form $v = const$, a naive representation as a set of logical orderings is problematic. This led Simmen et al. to introduce a more concise representation, which is discussed in the related work section. Note that Simmen's technique is not easily applicable to groupings, and no algorithm was proposed to efficiently maintain the set of available groupings. The order optimization component described here closes this gap by supporting both orderings and groupings. The problem of quadatic growth is avoided by only implicitly representing the set.

24.3 Overview

As we have seen, explicit maintenance of the set of logical orderings and groupings can be very expensive. However, the ADT `OrderingGrouping` required for plan generation does not need to offer access to this set: It only allows to test if a given interesting order or grouping is in the set and changes the set according to new functional dependencies. Hence, it is *not* required to explicitly represent this set; an implicit representation is sufficient as long as the ADT operations can be implemented atop of it. In other words, we need not be able to reconstruct the set of logical orderings and groupings from the state of the ADT. This gives us room for optimizations.

The initial idea (see [618]) was to represent sets of logical orderings as *states* of a *finite state machine* (FSM). Roughly, a state of the FSM represents a current physical ordering and the set of logical orderings that can be inferred from it given a set of functional dependencies. The edges (transitions) in the FSM are labeled by sets of functional dependencies. They lead from one state to another, if the target state of the edge represents the set of logical orderings that can be derived from the orderings the edge's source node represents by applying the set of functional dependencies the edge is labeled with. We have to use sets of functional dependencies, since a single

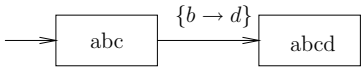


Figure 24.3: Possible FSM for groupings

algebraic operator may introduce more than one functional dependency.

Let us illustrate the idea by a simple example and then discuss some problems. In Figure 24.2, an FSM for the interesting order (a, b, c) and its prefixes (remember that we need prefix closure) and the set of functional dependencies $\{b \rightarrow d\}$ is given. When a physical ordering satisfies (a, b, c), it also satisfies its prefixes (a, b) and (a). This is indicated by the ϵ transitions. The functional dependency $b \rightarrow d$ allows to derive the *logical* orderings (a, b, c, d) and (a, b, d, c). This is handled by assuming that the *physical* ordering changes to either (a, b, c, d) or (a, b, d, c). Hence, these states have to be added to the FSM. We further add the transitions induced by $\{b \rightarrow d\}$. Note that the resulting FSM is a *non-deterministic finite state machine* (NFSM).

Assume we have an NFSM as above. Then (while ignoring groupings) the state of the ADT is a state of the NFSM and the operations of the ADT can easily be mapped to the FSM. Testing for a logical ordering can be performed by checking if the node with the ordering is reachable from the current state by following ϵ edges. If the set must be changed because of a functional dependency the state is changed by following the edge labeled with the functional dependency. Of course, the non-determinism is in our way.

While remembering only the active state of the NFSM avoids the problem of maintaining a set of orderings, the NFSM is not really useful from a practical point of view, since the transitions are non-deterministic. Nevertheless, the NFSM can be considered as a special *non-deterministic finite automaton* (NFA), which consumes the functional dependencies and "recognizes" the possible physical orderings. Further, an NFA can be converted into a *deterministic finite automaton* (DFA), which can be handled efficiently. Remember that the construction is based on the power set of the NFA's states. That is, the states of the DFA are sets of states of the NFA [538]. We do not take the deviation over the finite automaton but instead lift the construction of deterministic finite automatons from non-deterministic ones to finite state machines. Since this is not a traditional conversion, we give a proof of this step in Section 24.5.

Yet another problem is that the conversion from an NFSM to a *deterministic FSM* (DFSM) can be expensive for large NFSMs. Therefore, reducing the size of the NFSM is another problem we look at. We introduce techniques for reducing the set of functional dependencies that have to be considered and further techniques to prune the NFSM in Section 24.4.7.

The idea of a combined framework for orderings and groupings was presented in [617]. Here, the main point is to construct a similar FSM for groupings and integrate it into the FSM for orderings, thus handling orderings and groupings at the same time. An example of this is shown in Figure 24.3. Here, the FSM for the grouping $\{a, b, c\}$ and the functional dependency $b \rightarrow c$ is shown. We represent states for orderings as rounded boxes and states for groupings as rectangles. Note that although the FSM for groupings has a start node similar to the FSM for orderings, it is much smaller. This is due to the fact that groupings are only compatible with themselves, no nodes for

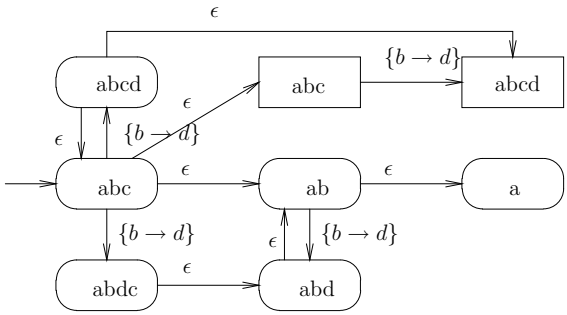


Figure 24.4: Combined FSM for orderings and groupings

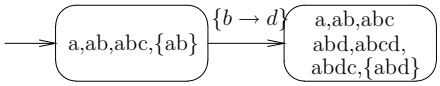


Figure 24.5: Possible DFSM for Figure 24.4

prefixes are required. However, the FSM is still non-deterministic: given the functional dependency $b \rightarrow c$, the grouping $\{a, b, c, d\}$ is compatible with $\{a, b, c, d\}$ itself and with $\{a, b, c\}$; therefore, there exists an (implicit) edge from each grouping to itself.

The FSM for groupings is integrated into the FSM for orderings by adding ϵ edges from each ordering to the grouping with the same attributes; this is due to the fact that every ordering is also a grouping. Note that although the ordering (a, b, c, d) also implies the grouping $\{a, b, c\}$, no edge is required for this, since there exists an ϵ edge to (a, b, c) and from there to $\{a, b, c\}$.

After constructing a combined FSM as described above, the full ADT supporting both orderings and groupings can easily be mapped to the FSM: The state of the ADT is a state of the FSM and testing for a logical ordering or grouping can be performed by checking if the node with the ordering or grouping is reachable from the current state by following ϵ edges (as we will see, this can be precomputed to yield the O(1) time bound for the ADT operations). If the state of the ADT must be changed because of functional dependencies, the state in the FSM is changed by following the edge labeled with the functional dependency.

However, the non-determinism of this transition is a problem. Therefore, for practical purposes the NFSM must be converted into a DFSM. The resulting DFSM is shown in Figure 24.5. Note that although in this simple example the DFSM is very small, the conversion could lead to exponential growth. Therefore, additional pruning techniques for groupings are presented in Section 24.4.7. However, the inclusion of groupings is not critical for the conversion, as the grouping part of the NFSM is nearly independent of the ordering part. In Section 24.6 we look at the size increase due to groupings. The memory consumption usually increases by a factor of two, which is the minimum expected increase, since every ordering is a grouping.

Some operators, like *sort*, change the physical ordering. In the NFSM, this is handled by changing the state to the node corresponding to the new physical ordering. Implied by its construction, in the DFSM this new physical ordering typically occurs in several nodes. For example, (a, b, c) occurs in both nodes of the DFSM in Figure 24.5. It is, therefore, not obvious which node to choose. We will take care of this problem

1. Determine the input
   (a) Determine interesting orders
   (b) Determine interesting groupings
   (c) Determine set of functional dependencies
2. Construct the NFSM
   (a) Construct states of the NFSM
   (b) Filter functional dependencies
   (c) Build filters for orderings and groupings
   (d) Add edges to the NFSM
   (e) Prune the NFSM
   (f) Add artificial start state and edges
3. Convert the NFSM into a DFSM
4. Precompute values
   (a) Precompute the compatibility matrix
   (b) Precompute the transition table

Figure 24.6: Preparation steps of the algorithm

during the construction of the NFSM (see Section 24.4.3).

24.4 Detailed Algorithm

24.4.1 Overview

Our approach consists of two phases. The first phase is the preparation step taking place before the actual plan generation starts. The output of this phase are the precomputed values used to implement the ADT. Then the ADT is used during the second phase where the actual plan generation takes place. The first phase is performed exactly once and is quite involved. Most of this section covers the first phase. Only Section 24.4.6 deals with the ADT implementation.

Figure 24.6 gives an overview of the preparation phase. It is divided into four major steps, which are discussed in the following subsections. Subsection 24.4.2 briefly reviews how the input to the first phase is determined and, more importantly, what it looks like. Section 24.4.3 describes in detail the construction of the NFSM from the input. The conversion from the NFSM to the DFSM is only briefly sketched in Section 24.4.4, for details see [538]. From the DFSM some values are precomputed which are then used for the efficient implementation of the ADT. The precomputation is described in Section 24.4.5, while their utilization and the ADT implementation are the topic of Section 24.4.6. Section 24.4.7 contains some important techniques to

reduce the size of the NFSM. They are applied in Steps 2 (b), 2 (c) and 2 (e). During the discussion, we illustrate the different steps by a simple running example. More complex examples can be found in Section 24.6.

24.4.2 Determining the Input

Since the preparation step is performed immediately before plan generation, it is assumed that the query optimizer has already determined which indices are applicable and which algebraic operators can possibly be used to construct the query execution plan.

Before constructing the NFSM, the set of interesting orders, the set of interesting groupings and the sets of functional dependencies for each algebraic operator are determined. We denote the set of sets of functional dependencies by $\mathcal{F}$. It is important for the correctness of our algorithms that we note which of the interesting orders are (1) produced by some algebraic operator or (2) only tested for. Note that the interesting orders which satisfy (1) may additionally be tested for as well. We denote those orderings under (1) by O_P, those under (2) by O_T. The total set of interesting orders is defined as $O_I = O_P \cup O_T$. The orders produced are treated slightly differently in the following steps. The groupings are classified similarly to the orderings: We denote the grouping produced by some algebraic operator by G_P, and those just tested for by G_T. The total set of interesting groupings is defined as $G_I = G_P \cup G_T$. More information on how to extract interesting groupings can be found in [876]. Furthermore, for a sample query the extraction of both interesting orders and groupings is illustrated in Section 24.6.

ToDo: details on determining interesting orders?

To illustrate subsequent steps, we assume that the set of sets of functional dependencies

$$\mathcal{F} = \{\{b \rightarrow c\}, \{b \rightarrow d\}\},$$

the interesting groupings

$$G_I = \{\{b\}\} \cup \{\{b, c\}\}$$

and the interesting orders

$$O_I = \{(b), (a, b)\} \cup \{(a, b, c)\}$$

have been extracted from the query. We assume that those in $O_T = \{(a, b, c)\}$ and $G_T = \{\{b, c\}\}$ are tested for but not produced by any operator, whereas those in $O_P = \{(b), (a, b)\}$ and $G_P = \{\{b\}\}$ may be produced by some algebraic operators.

24.4.3 Constructing the NFSM

An NFSM consists of a tuple (Σ, Q, D, q_o), where

- Σ is the input alphabet,
- Q is the set of possible states,
- $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, and
- q_0 is the initial state.

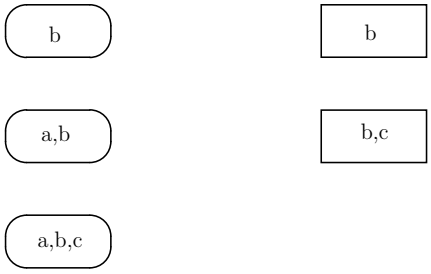


Figure 24.7: Initial NFSM for sample query

Coarsely, Σ consists of the functional dependencies, Q of the relevant orderings and groupings, and D describes how the orderings or groupings change under a given functional dependency. Some refinements are needed to provide efficient ADT operations. The details of the construction are described now.

For the order optimization part the states are partitioned in $Q = Q_I \cup Q_A \cup \{q_0\}$, where q_0 is an artificial state to initialize the ADT, Q_I is the set of states corresponding to interesting orderings and Q_A is a set of artificial states only required for the algorithm itself. Q_A is described later. Furthermore, the set Q_I is partitioned in Q_I^P and Q_I^T, representing the orderings in O_P and O_T, respectively. To support groupings, we add to Q_I^P states corresponding to the groupings in G_P and to Q_I^T states corresponding to the groupings in G_T.

The initial NFSM contains the states Q_I of interesting groupings and orderings. For the example, this initial construction not including the start state q_o is shown in Figure 24.7. The states representing groupings are drawn as rectangles and the states representing orderings are drawn with rounded corners.

When considering functional dependencies, additional groupings and orderings can occur. These are not directly relevant for the query, but have to be represented by states to handle transitive changes. Since they have no direct connection to the query, these states are called artificial states. Starting with the initial states Q_I, artificial states are constructed by considering functional dependencies

$$Q_A = (\Omega(O_I, \mathcal{F}) \setminus O_I) \cup (\Omega(G_I, \mathcal{F}) \setminus G_I).$$

In our example, this creates the states (b, c) and (a), as (b, c) can be inferred from (b) when considering $\{b \rightarrow c\}$ and (a) can be inferred from (a, b), since (a) is a prefix of (a, b). The result is show in Figure 24.8 (ignore the edges).

Sometimes the ADT has to be explicitly initialized with a certain ordering or grouping (e.g. after a `sort`). To support this, artificial edges are added later on. These point to the requested ordering or grouping (states in Q_I^P) and are labeled with the state that they lead to. Therefore, the input alphabet Σ consists of the sets of functional dependencies and produced orderings and groupings:

$$\Sigma = \mathcal{F} \cup Q_I^P \cup \{\epsilon\}.$$

In our example, $\Sigma = \{\{b \rightarrow c\}, \{b \rightarrow d\}, (b), (a, b), \{b\}\}$.

Accordingly, the domain of the transition relation D is

$$\begin{aligned} D \subseteq \quad & ((Q \setminus \{q_0\}) \times (\mathcal{F} \cup \{\epsilon\}) \times (Q \setminus \{q_0\})) \\ \cup \quad & (\{q_o\} \times Q_I^P \times Q_I^P). \end{aligned}$$

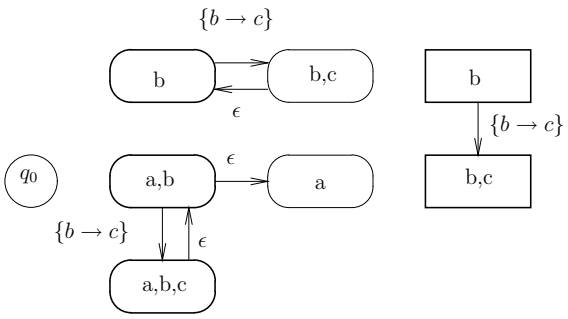


Figure 24.8: NFSM after adding D_{FD} edges

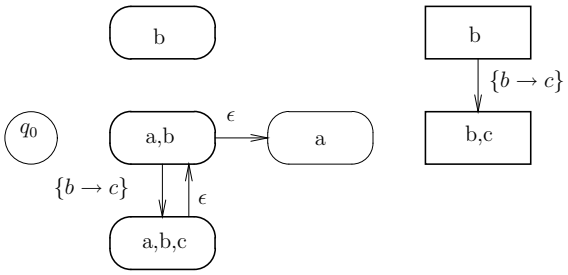


Figure 24.9: NFSM after pruning artificial states

The edges are formed by the functional dependencies and the artificial edges. Furthermore, ϵ edges exist between orderings and the corresponding groupings, as orderings are a special case of grouping:

$$\begin{aligned} D_{FD} &= \{(q, f, q') \mid q \in Q, f \in \mathcal{F} \cup \{\epsilon\}, q' \in Q, q \vdash_f q'\} \\ D_A &= \{(q_0, q, q) \mid q \in Q_I^P\} \\ D_{OG} &= \{(o, \epsilon, g) \mid o \in \Omega(O_I, \mathcal{F}), g \in \Omega(G_I, \mathcal{F}), o \equiv g\} \\ D &= D_{FD} \cup D_A \cup D_{OG} \end{aligned}$$

First, the edges corresponding to functional dependencies are added (D_{FD}). In our example, this results in the NFSM shown in Figure 24.8.

Note that the functional dependency $b \rightarrow d$ has been pruned, since d does not occur in any interesting order or grouping. The NFSM can be further simplified by pruning the artificial state (b, c), which cannot lead to a new interesting order. The result is shown in Figure 24.9. A detailed description of these pruning techniques can be found in Section 24.4.7.

The artificial start state q_0 has emanating edges incident to all states representing interesting orders in O_I^P and interesting groupings in G_I^P (D_A). Also, the states representing orderings have edges to their corresponding grouping states (D_{OG}), as every ordering is also a grouping. The final NFSM for the example is shown in Figure 24.10. Note that the states representing (a, b, c) and $\{b, c\}$ are not linked by an artificial edge since it is only tested for, as they are in Q_I^T.

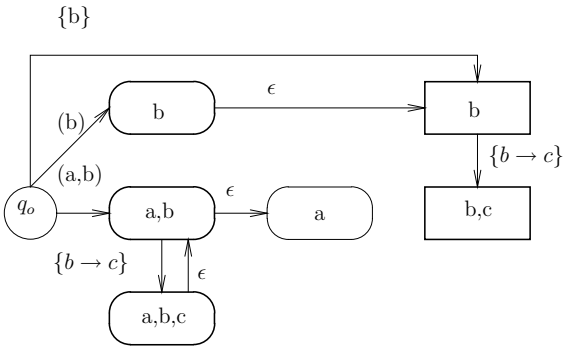


Figure 24.10: Final NFSM

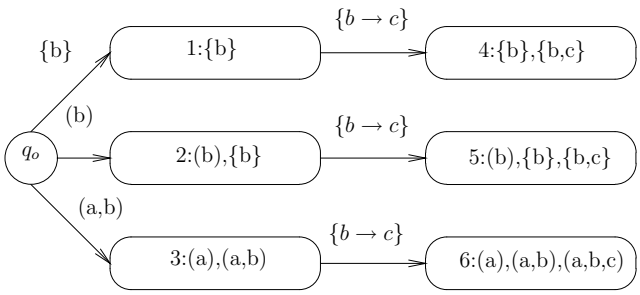


Figure 24.11: Resulting DFSM

24.4.4 Constructing the DFSM

The construction of the DFSM from the NFSM follows the standard power set construction that is used to translate an NFA into a DFA [538]. A formal description and a proof of correctness is given in Section 24.5. It is important to note that this construction preserves the start state and the artificial edges. The resulting DFSM for the example is shown in Figure 24.11.

24.4.5 Precomputing Values

To allow for an efficient precomputation of values, every occurrence of an interesting order, interesting grouping or set of functional dependencies is replaced by integers.

state	1: (a)	2: (a,b)	3: (a,b,c)	4: (b)	5: {b}	6: {b,c}
1	0	0	0	0	1	0
2	0	0	0	1	1	0
3	1	1	0	0	0	0
4	0	0	0	0	1	1
5	0	0	0	1	1	1
6	1	1	1	0	0	0

Figure 24.12: *contains* Matrix

state	1: $\{b \rightarrow c\}$	2: (a, b)	3: (b)	4: $\{b\}$
q_o	-	3	2	1
1	4	-	-	-
2	5	-	-	-
3	6	-	-	-
4	4	-	-	-
5	5	-	-	-
6	6	-	-	-

Figure 24.13: *transition* Matrix

This allows comparisons in constant time (equivalent entries are mapped to same integer). Further, the DFSM is represented by an adjacency matrix.

The precomputation step itself computes two matrices. The first matrix denotes whether an NFSM state in Q_I is active, i.e. an interesting order or an interesting grouping, is contained in a specific DFSM state. This matrix can be represented as a compact bit vector, allowing tests in O(1). For our running example, it is given (in a more readable form) in Figure 24.12. The second matrix contains the transition table for the DFSM relation D. Using it, edges in the DFSM can be followed in $O(1)$. For the example, the transition matrix is given in Figure 24.13.

24.4.6 During Plan Generation

During plan generation, larger plans are constructed by adding algebraic operators to existing (sub-)plans. Each subplan contains the available orderings and groupings in the form of the corresponding DFSM state. Hence, the state of the DFSM, a simple integer, is the state of our ADT `OrderingGrouping`.

When applying an operator to subplans, the ordering and grouping requirements are tested by checking whether the DFSM state of the subplan contains the required ordering or grouping of the operator. This is done by a simple lookup in the *contains* matrix.

If the operator introduces a new set of functional dependencies, the new state of the ADT is computed by following the according edge in the DFSM. This is performed by a quick lookup in the *transition* matrix.

For "atomic" subplans like table or index scans, the ordering and grouping is determined explicitly by the operator. The state of the DFSM is determined by a lookup in the transition matrix with start state q_o and the edge annotated by the produced ordering or grouping. For sort and group-by operators the state of the DFSM is determined as before by following the artificial edge for the produced ordering or grouping and then reapplying the set of functional dependencies that currently hold.

In the example, a sort on (b) results in a subplan with ordering/grouping state 2 (the state 2 is active in the DFSM), which satisfies the ordering (b) and the grouping $\{b\}$. After applying an operator which induces $b \rightarrow c$, the ordering/grouping changes to state 5 which also satisfies $\{b, c\}$.

24.4.7 Reducing the Size of the NFSM

Reducing the size of the NFSM is important for two reasons: First, it reduces the amount of work needed during the preparation step, especially the conversion from NFSM to DFSM. Even more important is that a reduced NFSM results in a smaller DFSM. This is crucial for plan generation, since it reduces the search space: Plans can only be compared and pruned if they have comparable ordering and a comparable set of functional dependencies (see [788, 789] for details). Reducing the size of the DFSM removes information that is not relevant for plan generation and, therefore, allows a more aggressive pruning of plans.

At first, the functional dependencies are pruned. Here, functional dependencies which can never lead to a new interesting order or grouping are removed. For convenience, we extend the definition of $\Omega(O, F)$ and define

$$\Omega(O, \epsilon) \quad := \quad \Omega(O, \emptyset).$$

Then the set of prunable functional dependencies F_P can be described by

$$\begin{aligned} \Omega_N(o, f) &:= \Omega(\{o\}, \{f\}) \setminus \Omega(\{o\}, \epsilon) \\ F_P &:= \{f | f \in F \wedge \forall o \in O_I \cup G_I : \\ &\quad (\Omega(\Omega_N(o, f), F) \setminus \Omega(\{o\}, \epsilon)) \cap (O_I \cup G_I) = \emptyset\}. \end{aligned}$$

Pruning functional dependencies is especially useful, since it also prunes artificial states that would be created because of the dependencies. In the example, this removed the functional dependency $b \rightarrow d$, since d does not appear in any interesting order or grouping. This step also removes the artificial states containing d.

The artificial states are required to build the NFSM, but they are not visible outside the NFSM. Therefore, they can be pruned and merged without affecting plan generation. Two heuristics are used to reduce the set of artificial states:

1. All artificial nodes that behave exactly the same (that is, their edges lead to the same states given the same input) are merged and
2. all edges to artificial states that can reach states in Q_I only through ϵ edges are replaced with corresponding edges to the states in Q_I.

More formally, the following pairs of states can be merged:

$$\begin{aligned} \{(o_1, o_2) \quad | \quad & o_1 \in Q_A, o_2 \in Q_A \wedge \forall f \in F : \\ & (\Omega(\{o_1\}, \{f\}) \setminus \Omega(\{o_1\}, \epsilon)) = \\ & (\Omega(\{o_2\}, \{f\}) \setminus \Omega(\{o_2\}, \epsilon))\}. \end{aligned}$$

The following states can be replaced with the next state reachable by an ϵ edge:

$$\begin{aligned} \{o \quad | \quad & o \in Q_A \wedge \forall f \in F : \\ & \Omega(\Omega(\{o\}, \epsilon), \{f\}) \setminus \{o\} = \\ & \Omega(\Omega(\{o\}, \epsilon) \setminus \{o\}, \{f\})\}. \end{aligned}$$

In the example, this removed the state (b, c), which was artificial and only led to the state (b).

These techniques reduce the size of the NFSM, but still most states are artificial states, i.e. they are only created because they can be reached by considering functional dependencies when a certain ordering or grouping is available. But many of these states are not relevant for the actual query processing. For example, given a set of interesting orders which consists only of a single ordering (a) and a set of functional dependencies which consists only of $a \rightarrow b$, the NFSM will contain (among others) two states: (a) and (a, b). The state (a, b) is created since it can be reached from (a) by considering the functional dependency, however, it is irrelevant for the plan generation, since (a, b) is not an interesting order and is never created nor tested for. Actually, in the example above, the whole functional dependency would be pruned (since *b* never occurs in an interesting order), but the problem remains for combinations of interesting orders: Given the interesting orders (a), (b) and (c) and the functional dependencies $\{a \rightarrow b, b \rightarrow a, b \rightarrow c, c \rightarrow b\}$, the NFSM will contain states for all permutations of *a*, *b* and *c*. But these states are completely useless, since all interesting orders consist only of a single attribute and, therefore, only the first entry of an ordering is ever tested.

Ideally, the NFSM should only contain states which are relevant for the query; since this is difficult to ensure, a heuristic can be used which greatly reduces the size of the NFSM and still guarantees that all relevant states are available: When considering a functional dependency of the form $a \rightarrow b$ and an ordering $o_1, o_2, \ldots, o_n$ with $o_i = a$ for some i ($1 \leq i \leq n$), the b can be inserted at any position j with $i < j \leq n + 1$ (for the special case of a condition $a = b$, $i = j$ is also possible). So, an entry of an ordering can only affect entries on the right of its own position. This means that it is unnecessary to consider those parts of an ordering which are behind the length of the longest interesting order; since that part cannot influence any entries relevant for plan generation, it can be omitted. Therefore, the orderings created by functional dependencies can be cut off after the maximum length of interesting orders, which results in less possible combinations and a smaller NFSM.

The space of possible orderings can be limited further by taking into account the prefix of the ordering: before inserting an entry b in an ordering $o_1, o_2, \ldots, o_n$ at the position i, check if there is actually an interesting order with the prefix $o_1, o_2, ...o_{i-1}, b$ and stop inserting if no interesting order is found. Also limit the new ordering to the length of the longest matching interesting order; further attributes will never be used. If functional dependencies of the form $a = b$ occur, they might influence the prefix of the ordering and the simple test described above is not sufficient. Therefore, a representative is chosen for each equivalence class created by these dependencies, and for the prefix test the attributes are replaced with their representatives. Since the set of interesting orders with a prefix of $o_1, \ldots, o_n$ is a superset of the set for the prefix $o_1, ...o_n, o_{n+1}$, this heuristic can be implemented very efficiently by iterating over i and reducing the set as needed.

Additional techniques can be used to avoid creating superfluous artifical states for groupings: First, in Step 2.3 (see Figure 24.6) the set of attributes occurring in interesting groupings is determined:

$$A_G = \{a \mid \exists g \in G_I : a \in g\}$$

Now, for every attribute a occurring on the right-hand side of a functional dependency

the set of potentially reachable relevant attributes is determined:

$$\begin{aligned} r(a,0) &= \{a\} \\ r(a,n) &= r(a,n-1) \cup \\ &\quad \{a' \mid \exists (a_1 \dots a_m \rightarrow a') \in \mathcal{F} : \\ &\quad\quad \{a_1 \dots a_m\} \cap r(a,n-1) \neq \emptyset\} \\ r(a) &= r(a, |\mathcal{F}|) \cap A_G \end{aligned}$$

This can be used to determine if a functional dependency actually adds useful attributes. Given a functional dependency $a_1 \dots a_n \rightarrow a$ and a grouping g with $\{a_1 \dots a_n\} \subseteq g$, a should only be added to g if $r(a) \not\subseteq g$, i.e. the attribute might actually lead to a new interesting grouping. For example, given the interesting groupings $\{a\}, \{a,b\}$ and the functional dependencies $a \rightarrow c, a \rightarrow d, d = b$. When considering the grouping $\{a\}$, the functional dependency $a \rightarrow c$ can be ignored, as it can only produce the attribute c, which does not occur in an interesting grouping. However, the functional dependency $a \rightarrow d$ should be added, since transitively the attribute b can be produced, which does occur in an interesting grouping.

Since there are no ϵ edges between groupings, i.e. groupings are not compatible with each other, a grouping can only be relevant for the query if it is a subset of an interesting ordering (as further attributes could be added by functional dependencies). However, a simple subset test is not sufficient, as equations of the form $a = b$ are also supported; these can effectively rename attributes, resulting in a slightly more complicated test:

In Step 2.3 (see Figure 24.6) the equivalence classes induced by the equations in $\mathcal{F}$ are determined and for each class a representative is chosen (a and $a_1 \dots a_n$ are attributes occuring in the G_I):

$$\begin{aligned} E(a,0) &= \{a\} \\ E(a,n) &= E(a,n-1) \cup \\ &\quad \{a' \mid ((a = a') \in \mathcal{F}) \vee ((a' = a) \in \mathcal{F})\} \\ E(a) &= E(a, |\mathcal{F}|) \\ e(a) &= \text{a representative choosen from } E(A) \\ e(\{a_1 \dots a_n\}) &= \{e(a_1) \dots e(a_n)\}. \end{aligned}$$

Using these equivalence classes, a mapped set of interesting groupings is produced that will be used to test if a grouping is relevant:

$$G_I^E = \{e(g) \mid g \in G_I\}$$

Now a grouping g can be pruned if $\nexists g' \in G_I^E : e(g) \subseteq g'$. For example, given the interesting grouping $\{a\}$ and the equations $a = b, b = c$, the grouping $\{d\}$ can be pruned, as it will never lead to an interesting grouping; however, the groupings $\{b\}$ and $\{c\}$ have to be kept, as they could change to an interesting grouping later on.

Note that although they appear to test similar conditions, the first pruning technique (using $r(a)$) is not dominated by the second one (using $e(a)$). Consider e.g. the

interesting grouping $\{a\}$, the equation $a = b$ and the functional dependency $a \rightarrow b$. Using only the second technique, the grouping $\{a, b\}$ would be created, although it is not relevant.

24.4.8 Complex Ordering Requirements

Specifying the ordering requirements of an operator can be surprisingly difficult. Consider the following SQL query:

```
select *
from   S s, R r
where  r.a=s.a and r.b=s.b and
       r.c=s.c and r.d=s.d
```

When answering this query using a sort-merge join, the operator has to request a certain odering. But there are many orderings that could be used: The intuitive ordering would be $abcd$, but $adcb$ or any other premutation could have been used as well. This is problematic, as checking for an exponential number of possibilities is not acceptable in general. Note that this problem is not specific to our approach, the same is true, e.g., for Simmen's approach.

The problem can be solved by defining a total ordering between the attributes, such that a canonical ordering can be constructed. We give some rules how to derive such an ordering below, but it can happen that such an ordering is unavailable (or rather the construction rules are ambiguous). Given, for example, two indices, one on $abcd$ and one on $adcb$, both orderings would be a reasonable choice. If this happens, the operators have two choices: Either they accept all reasonable orderings (which could still be an exponential number, but most likely only a few orderings remaing) or they limit themselves to one ordering, which could induce unnecessary sort operators. Probably the second choice is preferable, as the ambiguous case should be rare and does not justify the complex logic of the first solution.

The attribute ordering can be derived by using the following heuristical rules:

1. Only attributes that occur in sets without natural ordering (i.e. complex join predicates or grouping attributes) have to be ordered.
2. Orderings that are given (e.g., indices, user-requested orderings etc.) order some attributes.
3. Small orderings should be considered first. If an operator requires an ordering with the attributes abc, and another operator requires an ordering with the attributes bc, the attributes b and c should come before a.
4. The attributes should be ordered according to equivalence classes. If a is ordered before b, all orderings in $E(a)$ should be ordered before all orderings in $E(b)$.
5. Attributes should be ordered according to the functional dependencies, i.e. if $a \rightarrow b$, a should come before b. Note that $a = b$ suggests no ordering between a and b.
6. The remaining unordered attributes can be ordered in an arbitrary way.

The rules must check if they create contradictions. If this happens. the contradicting ordering must be omitted, resulting in potentially superfluous sort operators. Note that in some cases these sort operators are simply unavoidable: If for the example query one index on R exists with the ordering $abcd$ and one index on S with the ordering $dcba$, the heuristical rules detect a contradiction and choose one of the orderings. This results in a sort operator before the (sort-merge) join, but this sort could not have been avoided anyway.

24.5 Converting a NFSM into a DFSM

The algorithm described in this chapter first constructs a non-deterministic FSM and converts it to a deterministic FSM. For this conversion, the NFSM is treated like an NFA which is converted to a DFA. It has to be shown that the DFSM resulting from the conversion is equivalent to the initial NFSM:

24.5.1 Definitions

An NFA [538] consists of a tuple (Σ, Q, D, q_o, F), where Σ is the input alphabet, Q the set of possible states, $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the transition relation, q_0 the initial state and F the set of accepting states. All nodes reachable from a given set of nodes Q by following ϵ edges can be described by

$$\begin{aligned}\mathcal{E}_D^0(Q) &= Q \\ \mathcal{E}_D^i(Q) &= \{q' | \exists q \in \mathcal{E}_D^{i-1}(Q), (q, \epsilon, q') \in D\} \\ \mathcal{E}_D(Q) &= \bigcup_{i=0}^{\infty} \mathcal{E}_D^i(Q)\end{aligned}$$

Then the NFA *accepts* an input $w = w_1 w_2 ... w_n \in \Sigma^*$ if $S_n \cap F \neq \emptyset$ where

$$\begin{aligned}S_0 &= \mathcal{E}_D(q_o) \\ S_i &= \mathcal{E}_D(\{q' | \exists q \in S_{i-1} : (q, w_i, q') \in D\}).\end{aligned}$$

Similarly, a DFA [538] consists of a tuple $(\Sigma, Q, \Delta, q_o, F)$ where

$$\begin{aligned}& \Delta \subseteq Q \times \Sigma \times Q \\ \wedge \quad & \forall a, b, c \in Q, d \in \Sigma : \\ & ((a, d, b) \in \Delta \wedge (a, d, c) \in \Delta) \Rightarrow b = c.\end{aligned}$$

So a DFA is an NFA which only allows non-ambiguous non-ϵ transitions. The definition of accepting is analogous to the definition for NFAs.

An NFSM is basically an NFA without accepting states. It consists of a tuple (Σ, Q, D, q_o), where Σ is the input alphabet, Q the set of possible states, $D \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ the transition relation and q_0 the initial state. While an NFSM does not have any accepting states it is usually important to know which state is active after a given input, so in a way each state is accepting.

Likewise, a DFSM basically is a DFA without accepting states. It consists of a tuple (Σ, Q, Δ, q_o) where Σ, Q,Δ and q_o are analogous to the DFA. Again, while there is no set of accepting states, it is important to know which one is active after a given input.

24.5.2 The Transformation Algorithm

The commonly used algorithm to convert an NFA into a DFA (see [538]) can also be used to convert an NFSM into a DFSM. Since the accepting states are not required for the algorithm, the NFSM can be regarded as an NFA and converted into a "DFA", which is really a DFSM. The correctness of this transformation is shown in the next section.

The algorithm converts an NFSM (Σ, Q, D, q_o) in a DFSM $(\Sigma, Q', \Delta, q'_0)$ with $Q' \subseteq 2^Q$. It first constructs a start node $q'_0 = \mathcal{E}_D(\{q_0\})$ and then determines for all DFSM nodes q' all outgoing edges δ' by expanding all edges in the contained NFSM nodes:

$$\begin{aligned}\delta(q') &= \{(q', \sigma, q'_2 | \sigma \in \Sigma, q'_2 \neq \emptyset, \\ &\quad q'_2 = \{\mathcal{E}_D(q_2) | (q, \sigma, q_2) \in D, q \in q'\}\}.\end{aligned}$$

This results in the DFSM $(\Sigma, Q', \Delta, q'_o)$ with

$$\begin{aligned}Q'_0 &= \{q'_0\} \\ Q'_i &= \bigcup_{q' \in Q'_{i-1}} \{q'_2 | \exists \sigma \in \Sigma : (q', \sigma, q'_2) \in \delta(q')\} \\ Q' &= \bigcup_{i=0}^{\infty} Q'_i \\ \Delta &= \bigcup_{q' \in Q'} \delta(q).\end{aligned}$$

24.5.3 Correctness of the FSM Transformation

Proposition: Given an NFSM (Σ, Q, D, q_o), the DFSM $(\Sigma, Q' \subseteq 2^Q, \Delta, q'_0)$ constructed by using the transformation algorithm for NFA to DFA described in [538] behaves exactly like the NFSM, i.e.

$$\begin{aligned}&1) \quad \forall w \in \Sigma^*, q \in Q, q_0 \xrightarrow{w} q \; \exists q' \in Q' : q'_0 \xrightarrow{w} q' \wedge q \in q' \\ &2) \quad \forall w \in \Sigma^*, q'_a \in Q', q'_b \in Q', q_a \in q'_a, q_b \in q'_b : \\ &\qquad (q_a \xrightarrow{w} q_b) \text{ iff } (q'_a \xrightarrow{w} q'_b)\end{aligned}$$

Proof: Proposition 1) trivially follows from the definition of the transformation algorithm, see the definition of δ' and Q' in Section 24.5.2.

The proof for proposition 2) can be derived from the proof in [538], Chapter 2.3: there, it is shown that for all $w \in \Sigma^*$, given a node q in the NFA and a node q' in the transformed DFA with $q \in q'$, a node f' in the DFA contains a node f in the NFA if and only if $q \xrightarrow{w} f$ and $q' \xrightarrow{w} f'$. Since the DFSM is constructed using the same algorithm, this results in proposition 2).

Therefore, the conversion algorithm used to convert an NFA into a DFA can be used to convert the NFSM describing the ordering transitions to a DFSM that behaves the same way as the NFSM.

n	#Edges	t (ms)	#Plans	t/plan	t (ms)	#Plans	t/plan	% t	% #Plans	%. t/plan
5	n-1	2	1541	1.29	1	1274	0.78	2.00	1.21	1.65
6	n-1	9	7692	1.17	2	5994	0.33	4.50	1.28	3.55
7	n-1	45	36195	1.24	12	26980	0.44	3.75	1.34	2.82
8	n-1	289	164192	1.76	74	116562	0.63	3.91	1.41	2.79
9	n-1	1741	734092	2.37	390	493594	0.79	4.46	1.49	3.00
10	n-1	11920	3284381	3.62	1984	2071035	0.95	6.01	1.59	3.81
5	n	4	3060	1.30	1	2051	0.48	4.00	1.49	2.71
6	n	21	14733	1.42	4	9213	0.43	5.25	1.60	3.30
7	n	98	64686	1.51	20	39734	0.50	4.90	1.63	3.02
8	n	583	272101	2.14	95	149451	0.63	6.14	1.82	3.40
9	n	4132	1204958	3.42	504	666087	0.75	8.20	1.81	4.56
10	n	26764	4928984	5.42	2024	2465646	0.82	13.22	2.00	6.61
5	n+1	12	5974	2.00	1	3016	0.33	12.00	1.98	6.06
6	n+1	69	26819	2.57	6	12759	0.47	11.50	2.10	5.47
7	n+1	370	119358	3.09	28	54121	0.51	13.21	2.21	6.06
8	n+1	2613	509895	5.12	145	208351	0.69	18.02	2.45	7.42
9	n+1	27765	2097842	13.23	631	827910	0.76	44.00	2.53	17.41
10	n+1	202832	7779662	26.07	3021	3400945	0.88	67.14	2.29	29.62

Figure 24.14: Plan generation for different join graphs, Simmen's algorithm (left) vs. our algorithm (middle)

24.6 Experimental Results

The framework described in this chapter solves two problems: First, it provides an efficient representenation for reasoning about orderings and second, it allows keeping track of orderings and groupings at the same time. Since these topics are treated separately in the related work, the experimental results are split in two sections: In Section 24.7 the framework is compared to another published framework while only considering orderings, and in Section 24.8 the influence of groupings is evaluated.

24.7 Total Impact

We now consider how order processing influences the time needed for plan generation. Therefore, we implemented both our algorithm and the algorithm proposed by Simmen et al. [788, 789] and integrated them into a bottom-up plan generator based on [553].

To get a fair comparison, we tuned Simmen's algorithm as much as possible. The most important measure was to cache results in order to eliminate repeated calls to the very expensive *reduce* operation. Second, since Simmen's algorithm requires dynamic memory, we implemented a specially tailored memory management. This alone gave us a speed up by a factor of three. We further tuned the algorithm by thoroughly profiling it until no more improvements were possible. For each order optimization framework the plan generator was recompiled to allow for as many compiler optimizations as possible. We also carefully observed that in all cases both order optimization algorithms produced the same optimal plan.

We first measured the plan generation times and memory usage for TPC-R Query 8. A detailed discussion of this query follows in Section 24.8, here we ignored the grouping properties to compare it with Simmen's algorithm. The result of this experiment is summarized in the following table. Since order optimization is tightly integrated with plan generation, it is impossible to exactly measure the time spent just for order optimization during plan generation. Hence, we decided to measure the impact of order optimization on the total plan generation time. This has the advantage that we can also (for the first time) measure the impact order optimization has on plan generation time. This is important since one could argue that we are optimizing a problem

with no significant impact on plan generation time, hence solving a non-problem. As we will see, this is definitely not the case.

In subsequent tables, we denote by *t(ms)* the total execution time for plan generation measured in milliseconds, by *#Plans* the total number of subplans generated, by *t/plan* the average time (in microseconds) needed to introduce one plan operator, i.e. the time to produce a single subplan, and by *Memory* the total memory (in KB) consumed by the order optimization algorithms.

	Simmen	Our algorithm
t (ms)	262	52
#Plans	200536	123954
t/plan (μs)	1.31	0.42
Memory (KB)	329	136

From these numbers, it becomes obvious that order optimization has a significant influence on total plan generation time. It may come as a surprise that fewer plans need to be generated by our approach. This is due to the fact that the (reduced) FSM only contains the information relevant to the query, resulting in fewer states. With Simmen's approach, the plan generator can only discard plans if the ordering is the same and the set of functional dependencies is equal (respectively a subset). It does not recognize that the additional information is not relevant for the query.

In order to show the influence of the query on the possible gains of our algorithm, we generated queries with 5-10 relations and a varying number of join predicates — that is, edges in the join graph. We always started from a chain query and then randomly added some edges. For small queries we averaged the results of 100 queries and averaged 10 queries for large queries. The results of the experiment can be found in Fig. 24.14. In the second column, we denote the number of edges in terms of the number of relations (n) given in the first column. The next six columns contain (1) the total time needed for plan generation (in ms), (2) the number of (sub-) plans generated, and (3) the time needed to generate a subplan (in μs), i.e. to add a single plan operator, for (a) Simmen's algorithm (columns 3-5) and our algorithm (columns 6-8). The total plan generation time includes building the DFSM when our algorithm is used. The last three columns contain the improvement factors for these three measures achieved by our algorithm. More specifically, column *% x* contains the result of dividing the x column of Simmen's algorithm by the corresponding x column entry of our algorithm.

Note that we are able to keep the plan generation time below one second in most cases and three seconds in the worst case, whereas when Simmen's algorithm is applied, plan generation time can be as high as 200 seconds. This observation leads to two important conclusions:

1. Order optimization has a significant impact on total plan generation time.

2. By using our algorithm, significant performance gains are possible.

For completeness, we also give the memory consumption during plan generation for the two order optimization algorithms (see Fig. 24.15). For our approach, we also give the sizes of the DFSM which are included in the total memory consumption. All memory sizes are in KB. As one can see, our approach consumes about half as much memory as Simmen's algorithm.

n	#Edges	Simmen	Our Algorithm	DFSM
5	n-1	14	10	2
6	n-1	44	28	2
7	n-1	123	77	2
8	n-1	383	241	3
9	n-1	1092	668	3
10	n-1	3307	1972	4
5	n	27	12	2
6	n	68	36	2
7	n	238	98	3
8	n	688	317	3
9	n	1854	855	4
10	n	5294	2266	4
5	n+1	53	15	2
6	n+1	146	49	3
7	n+1	404	118	3
8	n+1	1247	346	4
9	n+1	2641	1051	4
10	n+1	8736	3003	5

Figure 24.15: Memory consumption in KB for Figure 24.14

24.8 Influence of Groupings

Integrating groupings in the order optimization framework allows the plan generator to easily exploit groupings and, thus, produce better plans. However, order optimization itself might become prohibitively expensive by considering groupings. Therefore, we evaluated the costs of including groupings for different queries.

Since adding support for groupings has no effect on the runtime behavior of the plan generator (all operations are still one table lookup), we measured the runtime and the memory consumption of the preparation step both with and without considering groupings. When considering groupings, we treated each interesting ordering also as an interesting grouping, i.e. we assumed that a grouping-based (e.g. hash-based) operator was always available as an alternative. Since this is the worst-case scenario, it should give an upper bound for the additional costs. All experiments were performed on a 2.4 GHz Pentium IV, using the gcc 3.3.1.

To examine the impact for real queries, we choose a more complex query from the well-known TPC-R benchmark ([844], Query 8):

```
select
    o_year,
    sum(case when nation = '[NATION]'
        then volume
        else 0
    end) / sum(volume) as mkt_share
from
    (select
        extract(year from o_orderdate) as o_year,
        l_extendedprice * (1-l_discount) as volume,
```

```
    n2.n_name as nation
from  part,supplier,lineitem,orders,customer,
      nation n1,nation n2,region
where
    p_partkey = l_partkey and
    s_suppkey = l_suppkey and
    l_orderkey = o_orderkey and
    o_custkey = c_custkey and
    c_nationkey = n1.n_nationkey and
    n1.n_regionkey = r_regionkey and
    r_name = '[REGION]' and
    s_nationkey = n2.n_nationkey and
    o_orderdate between date '1995-01-01' and
        date '1996-12-31' and
    p_type = '[TYPE]'
) as all_nations
group by o_year
order by o_year;
```

When considering this query, all attributes used in joins, group-by and order-by clauses are added to the set of interesting orders. Since hash-based solutions are possible, they are also added to the set of interesting groupings. This results in the sets

$$
\begin{aligned}
O_I^P &= \{(o_year), (o_partkey), (p_partkey),\\
&\quad (l_partkey), (l_suppkey), (l_orderkey),\\
&\quad (o_orderkey), (o_custkey), (c_custkey),\\
&\quad (c_nationkey), (n1.n_nationkey),\\
&\quad (n2.n_nationkey), (n_regionkey),\\
&\quad (r_regionkey), (s_suppkey), (s_nationkey)\}\\
O_I^T &= \emptyset\\
G_I^P &= \{\{o_year\}, \{o_partkey\}, \{p_partkey\},\\
&\quad \{l_partkey\}, \{l_suppkey\}, \{l_orderkey\},\\
&\quad \{o_orderkey\}, \{o_custkey\}, \{c_custkey\},\\
&\quad \{c_nationkey\}, \{n1.n_nationkey\},\\
&\quad \{n2.n_nationkey\}, \{n_regionkey\},\\
&\quad \{r_regionkey\}, \{s_suppkey\}, \{s_nationkey\}\}\\
G_I^T &= \emptyset
\end{aligned}
$$

Note that here O_I^T and G_I^T are empty, as we assumed that each ordering and grouping would be produced if beneficial. For example, we might assume that it makes no sense to intentionally group by o_year: If a tuple stream is already grouped by o_year it makes sense to exploit this, however, instead of just grouping by o_year it could make sense to sort by o_year, as this is required anyway (although here it only makes sense if the sort operator performs early aggregation). In this case, $\{o_year\}$ would move from G_I^P to G_I^T, as it would be only tested for, but not produced.

The set of functional dependencies (and equations) contains all join conditions and constant conditions:

$$\begin{aligned}\mathcal{F} = \{&\{p_partkey = l_partkey\}, \{\emptyset \rightarrow p_type\},\\ &\{o_custkey = c_custkey\}, \{\emptyset \rightarrow r_name\},\\ &\{c_nationkey = n1.n_nationkey\},\\ &\{s_nationkey = n2.n_nationkey\},\\ &\{l_orderkey = o_orderkey\},\\ &\{s_suppkey = l_suppkey\},\\ &\{n1.n_regionkey = r_regionkey\}\}\end{aligned}$$

To measure the influence of groupings, the preparation step was executed twice: Once with the data as given above and once with $G_I^P = \emptyset$ (i.e. groupings were ignored). The space and time requirements are shown below:

	With Groups	Without Groups
Duration [ms]	0.6ms	0.3ms
DFSM [nodes]	63	32
Memory [KB]	5	2

Here time and space requirements both increase by a factor of two. Since all interesting orderings are also treated as interesting groupings, a factor of about two was expected.

While Query 8 is one of the more complex TPC-R queries, it is not overly complex when looking at order optimization. It contains 16 interesting orderings/groupings and 8 functional dependencies, but they cannot be combined in many reasonable ways, resulting in a comparatively small DFSM. In order to get more complex examples, we produced randomized queries with $5 - 10$ relations and a varying number of join predicates. We always started from a chain query and then randomly added additional edges to the join graph. The results are shown for $n - 1$, n and $n + 1$ additional edges. In the case of 10 relations, this means that the join graph consisted of 18, 19 and 20 edges, respectively.

The time and space requirements for the preparation step are shown in Figure 24.16 and Figure 24.17, respectively. For each number of relations, the requirements for the combined framework (`o+g`) and the framework ignoring groupings (`o`) are shown. The numbers in parentheses ($n - 1$, n and $n + 1$) are the number of additional edges in the join graph.

As with Query 8, the time and space requirements roughly increase by a factor of two when adding groupings. This is a very positive result, given that a factor of two can be estimated as a lower bound (since every interesting ordering is also an interesting grouping here). Furthermore, the absolute time and space requirements are very low (a few ms and a few KB), encouraging the inclusion of groupings in the order optimization framework.

24.9 Annotated Bibliography

Very few papers exist on order optimization. While the problem of optimizing interesting orders was already introduced by Selinger et al. [750], later papers usually

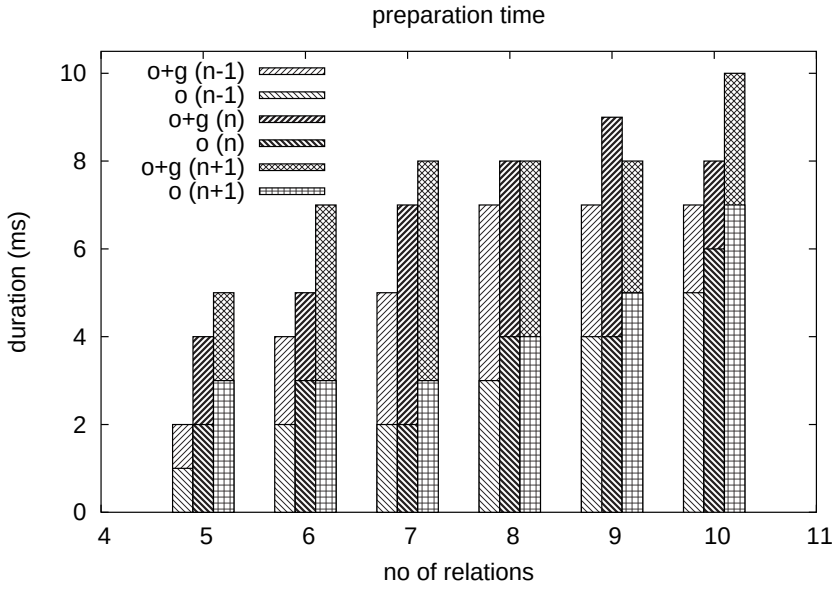


Figure 24.16: Time requirements for the preparation step

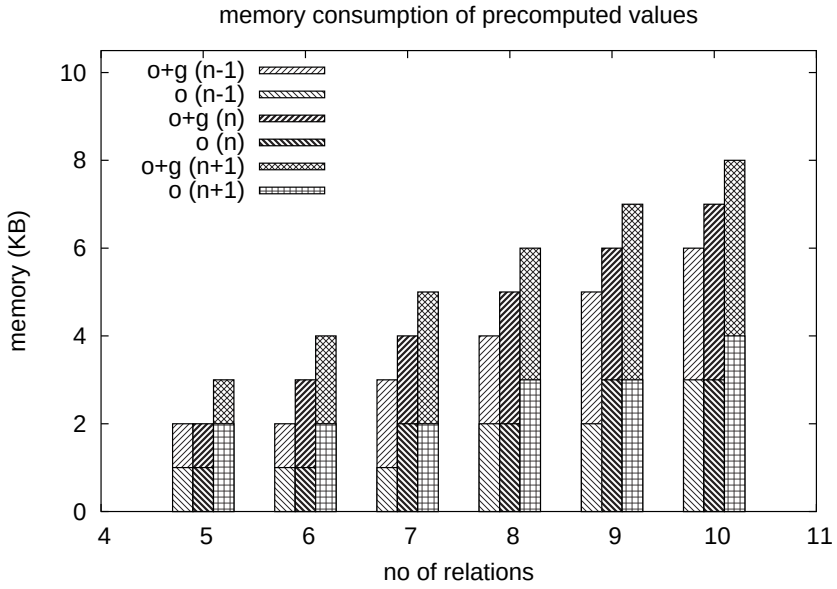


Figure 24.17: Space requirements for the preparation step

concentrated on exploiting, pushing down or combining orders, not on the abstract handling of orders during query optimization.

Papers by Simmen, Shekita, and Malkemus [788, 789] introduced a framework based on functional dependencies for reasoning about orderings. Since this is the only paper which really concentrates on the abstract handling orders and our approach is similar in the usage of functional dependencies, we will describe their approach in some more detail.

For a plan node they keep just a single (physical) ordering. Additionally, they associate all the applicable functional dependencies with a plan node. Hence, the lower-bound space requirement for this representation is essentially $\Omega(n)$, where n is the number of functional dependencies derived from the query. Note that the set of functional dependencies is still (typically) much smaller than the set of all logical orderings. In order to compute the function `containsOrdering`, Simmen et al. apply a *reduction algorithm* on both the ordering associated with a plan node and the ordering given as an argument to `containsOrdering`. Their reduction roughly does the opposite of deducing more orderings using functional dependencies. Let us briefly illustrate the reduction by an example. Assume the physical ordering a tuple stream satisfies is (a), and the required ordering is (a, b, c). Further assume that there are two functional dependencies available: $a \rightarrow b$ and $a, b \rightarrow c$. The reduction algorithm is performed on both orderings. Since (a) is already minimal, nothing changes. Let us now reduce (a, b, c). We apply the second functional dependency first. Using $a, b \rightarrow c$, the reduction algorithm yields (a, b) because c appears in (a, b, c) after a and b. Hence, c is removed. In general, every occurrence of an attribute on the right-hand side of a functional dependency is removed if all attributes of the left-hand side of the functional dependency precede the occurrence. Reduction of (a, b) by $a \rightarrow b$ yields (a). After both orderings are reduced, the algorithm tests whether the reduced required ordering is a prefix of the reduced physical ordering. Note that if we applied $a \rightarrow b$ first, then (a, b, c) would reduce to (a, c) and no further reduction would be possible. Hence, the rewrite system induced by their reduction process is not confluent. This problem is not mentioned by Simmen et al., but can have the effect that `containsOrdering` returns *false* whereas it should return *true*. The result is that some orderings remain unexploited; this could be avoided by maintaining a minimal set of functional dependencies, but the computation costs would probably be prohibitive. This problem does not occur with our approach. On the complexity side, every functional dependency has to be considered by the reduction algorithm at least once. Hence, the lower time bound is $\Omega(n)$.

In case all functional dependencies are introduced by a single plan node and all of them have to be inserted into the set of functional dependencies associated with that plan node, the lower bound for `inferNewLogicalOrderings` is also $\Omega(n)$.

Overall, Simmen et al. proposed the important framework for order optimization utilizing functional dependencies and nice algorithms to handle orderings during plan generation, but the space and time requirements are unfortunate since plan generation might generate millions of subplans. Also note that the reduction algorithm is not applicable for groupings (which, of course, was never intended by Simmen): Given the grouping $\{a, b, c\}$ and the functional dependencies $a \rightarrow b$ and $b \rightarrow c$, the grouping would be reduced to $\{a, c\}$ or to $\{a\}$, depending on the order in which the reductions are performed. This problem does not occur with orderings, as the attributes are sorted and can be reduced back to front.

A recent paper by Wang and Cherniack [876] presented the idea of combining order optimization with the optimization of groupings. Based upon Simmen's framework, they annotated each attribute in an ordering with the information whether it is actually ordered by or grouped by. For a single attribute a, they write $O_{a^O}(R)$ to denote that R is ordered by a, $O_{a^G}(R)$ to denote that R is grouped by a and $O_{a^O \rightarrow b^G}$ to denote that R is first ordered by a and then grouped by b (within blocks of the same

a value). Before checking if a required ordering or grouping is satisfied by a given plan, they use some inference rules to get all orderings and groupings satisfied by the plan. Basically, this is Simmen's reduction algorithm with two extra transformations for groupings. In their paper the check itself is just written as $\in$, however, at least one reduction on the required ordering would be needed for this to work (and even that would not be trivial, as the stated transformations on groupings are ambiguous). The promised details in the cited technical report are currently not available, as the report has not appeared yet. Also note that, as explained above, the reduction approach is fundamentally not suited for groupings. In Wang's and Cherniack's paper, this problem does not occur, as they only look at a very specialized kind of grouping: As stated in their Axiom 3.6, they assume that a grouping $O_{a^G \rightarrow b^G}$ is first grouped by a and then (within the block of tuples with the same a value) grouped by b. However, this is a very strong condition that is usually not satisfied by a hash-based grouping operator. Therefore, their work is not general enough to capture the full functionality offered by a state-of-the-art query execution engine.

In this chapter, we followed [618, 617].

Chapter 25

Other Issues in Plan Generation

25.1 Plan Generation for Compressed Databases

compressed data bases: [155]

25.2 Generating DAGs-Plans

generating DAGs:

```
@misc{ roy-optimization,
  author = "Prasan Roy",
  title = "Optimization of DAG-Structured Query Evaluation Plans",
  url = "citeseer.nj.nec.com/roy98optimization.html" }
```

Part VI

Cardinality and Cost Estimates

Chapter 26

Introduction

Every physical algebraic operator has associated costs. We distinguish between CPU-costs, I/O-costs, and (possibly) communication costs. To derive the total cost these cost factors are summed up, possibly weighted with some factors. These factors are used to emphasize CPU-costs for CPU bound systems and I/O costs for I/O bound systems. For example, we can compute the total costs by the formula *total cost* $=$ $w_{cpu} * \textit{CPU-cost} + w_{I/O} * \textit{I/O-cost}$ where the w_x are weighting factors. Given the costs for all operators occurring in a plan, the total cost of the plan is just the sum of the costs of the operators in the plan.

Operator costs depend on the cardinality of its input(s). For example, the cost of evaluating $\sigma_p(e_1)$ can be estimated by nc_p if n is the cardinality of the intermediate result of the plan e_1 and c_p is the average cost of evaluating the predicate p on a single input tuple. This is the CPU cost of the selection operator.

The problem of estimating cardinalities is discussed in Section **??**. A prerequisite for cardinality estimation are database statistics.

Database statistics and their usage to estimate cardinalities and costs often rely on certain assumptions, that do not necessarily hold in practice. These assumptions are [174, 570]:

- uniform distribution of attribute values
- indepencence of attribute values
- for a join it is assumed that the projection of the join attributes of the smaller relation are a subset of the join attributes of the larger relation (assumes inclusion dependency)
- constant number of tuples per page
- random placement of tuples among pages

These assumption form the starting point for every database statistics and estimation procedure. Implications of these assumptions are discussed in [570]. Elaborations to account for deviations from these assumptions are still a matter of research.

Database statistics are used to estimate cardinalities, sizes and costs during plan generation [570]. The core numbers that are kept for every relation/extent are cardinality and size in pages. If slotted pages are used, the number of overflow records is of interest, too. If possible, the CPU and I/O costs for scanning are sampled.

Additionally, for every attribute, the number of different values, min/max values and the average length of the values (in bytes) are collected.[1] These are used to estimate selectivities (see next chapter) by relying on a uniform distribution of values. However, this assumption often turns out to be wrong. Hence, other statistics are additionally collected. For example, for the 10 most/least frequently occurring values, their number of occurrences is collected. Another statistics that is often collected are *quantiles*. If, for example 10 quantiles are collected, then ten values are stored such that they equal or exceed 10%, 20%,. . . , 100% of the occurring values.

A more elaborate technique is to collect histograms [435, 671, 451, 576]. Most common are equi-width and equi-depth histograms [614]. For an equi-width histogram, the (attribute's active) domain is partitioned into intervals of equal width. For every interval, the number of occurring values is memorized. Within every interval, uniform distribution is assumed. In equi-depth histograms, the domain is partitioned such that in every partition the same number of values occurs.

Statistics should carry over to intermediate results. As we will see in the next section, they can be estimated given an operator (selection, join, etc.) and the statistics of the input(s). The cardinalities for the intermediate results are hold as cardinality properties in every plan operator node. The mostly used estimated statistics are:

1. min/max/avg size of a single intermediate result tuple
2. min/max cardinality proven (key, fd's)
3. min/max cardinality estimated
4. min/max distinct values proven
5. min/max distinct values estimated

For every IU (attribute) of a tuple in an intermediate result, we should be able to estimate the number of distinct values it is bound to and for collection-valued IUs we must estimate the sizes of the collections they are bound to.

26.1 Selection

After scanning a relation, we exactly know how many tuples are retrieved (if our statistics is correct). But how many tuples do we have after a selection, a join, a projection with duplicate elimination, or a grouping?

For selections, the approach can be as simple (and crude) as the following. Denote by the *selectivity* f (also called *filter factor*) of a selection operation the fraction of tuples that qualify, that is $f = \frac{\sigma_p(|R|)}{|R|}$. Then, f can be estimated for every class of predicates p. For example,

$$\begin{array}{lcll} f & = & 1/10 & \text{for equality predicates} \\ f & = & 1/3 & \text{for range predicates} \\ f & = & 1/20 & \text{for is-null predicates} \\ f & = & 1/10 & \text{for like predicates} \end{array}$$

[1] Since border values are often used to denote exceptional values, the second highest and second lowest value are used.

Obviously, this is very rough. At the other end of the spectrum, we are able to estimate the number of output tuples very precisely, if we have an equality predicate for a key attribute and a constant. This will always result in at most a single tuple. For more conditions restricting the output to a single tuple and their usefulness see [653, 787].

In general the statistics will be used to estimate the selectivity. For example, for an equality predicate p of the form $A = c$, for a constant c and an attribute A, we estimate the selectivity $f = 1/d_A$ if d_A is the number of distinct values for A. Hence, we estimate that $\sigma_p(R)$ returns $||R|| * 1/d_A$ tuples if $||R||$ is the number of tuples in R. If we have more than a single selection, we multiply the selectivities. We combine selectivity factors according the following formulas:

$$\begin{aligned} f_{p \wedge q} &= f_p * f_q \\ f_{p \vee q} &= f_p + f_q - f_p * f_q \\ f_{\neg p} &= 1 - f_p \end{aligned}$$

Note that this assumes indepencence. Since this is a very dangerous assumption, another approach sometimes found is to take the minimum of all selectivities as the selectivity of a conjunction of predicates.

26.2 Join

The selectivity of a join $R \bowtie S$ is defined as $f = \frac{||R \bowtie S||}{||R|| * ||S||}$. We can precisely determine the number of output tuples, if the join is an equijoin over foreign key attributes in S and key attributes in R. Then for every tuple found in S exactly one matching partner in R is found. Hence, $f = 1/||R||$ holds. To estimate the selectivity of a general n:m-join $R \bowtie_{R.A=S.B} S$ under an inclusion dependency, the formula $f = 1/\max(d_A, d_B)$ can be used, where d_A and d_B are the number of distinct values for attribute A and B, resp. [849, 750].

The classic [750] contains a table for computing selectivities for different predicates. This table contains the above mentioned estimates plus some more. With an adopted notation, we present the estimates in Table 26.1.

26.3 Projection, Grouping, and Duplicate Elimination

We come to estimate the size of a projection with duplicate elimination. Before we present the formula, note that this estimate also serves to estimate the result size of a grouping operation. There are several papers on estimating projection sizes [18, 317, 316, 608]. We now present the formula by Mukkamala and Jajodia [608] for estimating the result size of a projection. Let $R'(A, B)$ be a relation with attributes A and B with $|dom(A)| = m$ and $|dom(B)| = n$. Let p be the number of distinct A values for a given B value and let q be the number of distinct B values for a given A value. Further let Q be a unary relation of k distinct A values and let R be the natural join of Q and R'. If $np = mq$ then the expected number of tuples in the projection of R onto B is

$$E_s(R[B]) = n \left(1 - \frac{\binom{m-p}{k}}{\binom{m}{k}} \right)$$

$a = c$	$f = 1/d_a$ if d_a is known
	$f = 1/10$ otherwise
$a_1 = a_2$	$f = 1/\max(d_{a_1}, d_{a_2})$ if d_{a_1} and d_{a_2} are known
	$f = 1/d_{a_i}$ if only d_{a_i} ($i \in \{1, 2\}$) is known
	$f = 1/10$ otherwise
$a > c$	$f = \frac{\max(a)-a}{\max(a)-\min(a)}$ if a is of a number type and $\min(a)$ and $\max(a)$ are known
	$f = 1/3$ otherwise
a between c_1 and c_2	$f = \frac{c_2-c_1}{\max(a)-\min(a)}$ if $\min(a)$ and $\max(a)$ are known
	$f = 1/4$ otherwise
a in list	$f = \min(1/2, n * f)$ where n is the number of values in the list and f is the selectivity of $a = c$
a in subquery	$f = n/p$ where n is the (estimated) result cardinality of the subquery and p is the product of the cardinalities of all rthe relations in the subqueries **from** clause

We used the following notation:

a, a_1, a_2	attributes
c, c_1, c_2	constant
d_a	number of distinct values of a
$\max(a)$	maximum value of attribute a
$\min(a)$	minimum value of attribute a

Table 26.1: Selectivity estimates for several predicates [750]

A simpler upper bound can be derived by multiplying the number of distinct values of the attributes which are projected or which occur in the **group by** clause.

26.4 Feedback from Runtime

[41] [476] [153] [8] [806] [100] [463] [681] [852]

26.5 Bibliography

Database Statistics [254] Overview article of data reduction techniques. [18] [315]

cardinality of join result [702]

[28, 139, 435, 671, 576, 863, 614]

Cost Models

[8, 9] [53, 68, 77, 164, 126] [157] [167, 169, 170, 173, 174] [196] [177, 178, 221, 260, 297, 301, 302, 303, 377] [380, 401, 383, 382, 381] [437, 435] [504, 547, 548, 557, 560] [570, 561, 562, 576, 863, 586, 605, 608] [671, 670, 660, 766] [833] [225] [28, 451]

selectivity on strings: [450, 447, 504]

Two Cost Models for Join Algorithms [377]
XML: [7, 156, 279, 545, 667, 668, 665, 905, 906]

Chapter 27

Statistics and Cardinality Estimates

27.1 Uniformity and Independence Assumption

Database statistics are used to estimate cardinalities, sizes and costs during plan generation [570]. The core numbers that are kept for every relation/extent are cardinality and size in pages. If slotted pages are used, the number of overflow records is of interest, too. If possible, the CPU and I/O costs for scanning are sampled.

Additionally, for every attribute, the number of different values, min/max values and the average length of the values (in bytes) are collected.[1] These are used to estimate selectivities (see next chapter) by relying on a uniform distribution of values. However, this assumption often turns out to be wrong. Hence, other statistics are additionally collected. For example, for the 10 most/least frequently occurring values, their number of occurrences is collected. Another statistics that is often collected are *quantiles*. If, for example 10 quantiles are collected, then ten values are stored such that they equal or exceed 10%, 20%,..., 100% of the occurring values.

A more elaborate technique is to collect histograms [435, 671, 451, 576]. Most common are equi-width and equi-depth histograms [614]. For an equi-width histogram, the (attribute's active) domain is partitioned into intervals of equal width. For every interval, the number of occurring values is memorized. Within every interval, uniform distribution is assumed. In equi-depth histograms, the domain is partitioned such that in every partition the same number of values occurs.

Statistics should carry over to intermediate results. As we will see in the next section, they can be estimated given an operator (selection, join, etc.) and the statistics of the input(s). The cardinalities for the intermediate results are hold as cardinality properties in every plan operator node. The mostly used estimated statistics are:

1. min/max/avg size of a single intermediate result tuple
2. min/max cardinality proven (key, fd's)
3. min/max cardinality estimated

[1] Since border values are often used to denote exceptional values, the second highest and second lowest value are used.

4. min/max distinct values proven
5. min/max distinct values estimated

For every IU (attribute) of a tuple in an intermediate result, we should be able to estimate the number of distinct values it is bound to and for collection-valued IUs we must estimate the sizes of the collections they are bound to.

27.2 Dropping the Uniformity Assumption

27.2.1 ToDo

- [931, 932]
- sampling: [28]

27.3 Dropping the Independence Assumption

27.4 Bibliography

- String predicates: [135]

Cardinality estimates for

- [18], [319], [307, 306]
- Join [702] [168, 171, 172] [305], [309],[437],[434], [833]
- semijoin: [304]
- Projection: [304], [317], [316], [318],[608]
- Count(e), e arbitrary relational expression, sampling: [422]
- partial preaggregation: [416]
- Wang et al: [874, 873]

Maintenance: [322]
Streams: [367]
Joins: [468]

Chapter 28

Cost functions for selected algebraic operators

As we have seen, estimating the CPU-costs for selections is very simple. The same is true for the mapping operator as long as the costs of the subscripts (predicates and/or function calls) can be estimated. How to do this, especially for user-defined functions is still a matter of research. The only approach known so far is averaging a sample of executions.

28.1 Scan Operations

DiskModels

[845]

28.2 I/O costs for index-based access

If we scan a whole relation, estimating the I/O cost is rather trivial. This becomes different, if we use an index to retrieve RIDs and subsequently access the base relations. The question is how many page accesses occur. The solution to the question was given by Yao [920]. Define

$$\begin{aligned} n &= \text{number of records of the file} \\ p &= \text{number of pages of the file} \\ q &= \text{number of (distinct) records selected by the query} \end{aligned}$$

Under the assumption that $1 < p \leq n$ and $q \leq n - n/p$, we can estimate the number of accessed pages by

$$p * \left[1 - \prod_{i=1}^{q} \frac{n * D - i + 1}{n - i + 1} \right]$$

where $D = 1 - 1/p$. Yao's formula is an improvement of Cardena's formula [111]. A pretty precise approximation of Yaos's formula that can be computed much faster can

Notation	Name	Value	Comment
P	page size	8KB	
C_T	total I/O cost		
T_S	avg seek time	8.3ms	
T_L	avg latency time	2.6ms	= half disc rotation time
T_X	page transfer cost	2.6ms	derived from transfer rate of device
N_S	number of seeks		
$N_{I/O}$	number of I/Os		
N_X	number of transfers		
F	universal fudge factor	1.2	for storage overhead
$\|R\|$	size of relation (pages)		$\|R\| \leq \|S\|$
$\|S\|$	size of relation (pages)		$\|R\| \leq \|S\|$
M	memory size		
M_R	pages used for R		
M_S	pages used for S		$M_S = M - M_R$
I	input buffer size		
O	output buffer size		

Table 28.1: Parameter for cost model [377]

be found in [308]:

$$p * (1 - (1 - \frac{q}{fp})^f)$$

where $f = n/p$ is the blocking factor, that is the (avg) number of tuples per page.

The number computed with the Yao formula does not include the number of page accesses performed by the index. For a B-tree, the number of page accesses is equal to its height, typically three. For extensible hashing, one page access suffices.

28.3 I/O costs for join algorithms

Cost models that estimate the number of block accesses for different join algorithms can be found in [377, 401].

We briefly review the cost model of Hass, Carey, Livny, Shukla [377]. Table 28.1 summarizes the parameters needed for the cost model.

Total I/O Cost

$$C_T = N_S T_S + N_{I/O} T_L + N_X T_X \tag{28.1}$$

Comment: Small seeks from one cylinder to the next are ignored, since the transfer time (216ms) of a whole cylinder is much higher than the small seek time to jump to the next cylinder.

Blockwise nested loop join

$$
\begin{aligned}
N_S &= 2N_B && (28.2)\\
N_{I/O} &= N_B * (1 + \lceil \frac{|S|}{M_S} \rceil) && (28.3)\\
N_X &= |R| + (N_B * |S|) && (28.4)\\
N_B &= \lceil \frac{|R|F}{M_R} \rceil && (28.5)
\end{aligned}
$$

Sort-Merge Join assumption: $M > \sqrt{F|S|}$.

$$
\begin{aligned}
N_S &= 4 + \lceil \frac{|R|}{MPR} \rceil + \lceil \frac{|S|}{MPR} \rceil && (28.6)\\
N_{I/O} &= \lceil \frac{|R|}{I} \rceil + \lceil \frac{|R|}{O} \rceil + \lceil \frac{|S|}{I} \rceil + \lceil \frac{|S|}{O} \rceil + \lceil \frac{|R|}{MPR} \rceil + \lceil \frac{|S|}{MPR} \rceil && (28.7)\\
N_X &= 3|R| + 3|S| && (28.8)\\
RL &= \lceil \frac{2 * WS}{F} \rceil && (28.9)\\
MPR &= \frac{M}{NR_R + NR_S} && (28.10)\\
NR_R &= \lceil \frac{|R|}{RL} \rceil && (28.11)\\
NR_S &= \lceil \frac{|S|}{RL} \rceil && (28.12)\\
WS &= M - I - O && (28.13)
\end{aligned}
$$

where I input buffer size, O output buffer size, MPR is the merge buffer size. Additionally, for the sorting some workspace $WS = M - I - O$ must be allocated for array or turnament tree or something thelike. RL is the run length. NR_R is the number of runs for R, NR_S is the number of runs for S.

Simple Hash Join

$$
\begin{aligned}
N_S &= 2 * NI + 2 * (NI - 1) = 4 * NI - 2 && (28.14)\\
N_{I/O} &= \frac{1}{I}[NI * (|R| + |S|) - \frac{1}{2} NI * (NI - 1) * (K_R + K_S)] \\
&\quad + \frac{1}{O}[(NI - 1) * (|R| + |S|) - \frac{1}{2} NI * (NI - 1) * (K_R + K_S)] && (28.15)\\
N_X &= (2 * NI - 1) * (|R| + |S|) - NI * (NI - 1) * (K_R + K_S) && (28.16)\\
K_R &= \lceil \frac{WS}{F} \rceil && (28.17)\\
K_S &= \lfloor \frac{|S|K_R}{|R|} \rfloor && (28.18)\\
NI &= \lceil \frac{|R|F}{WS} \rceil && (28.19)\\
WS &= M - I - O && (28.20)
\end{aligned}
$$

NI	number of iterations
K_R	number of pages of R kept in memory on each iteration
K_S	number of pages of S that match the in-memory portion of R on each iteration

Grace Hash Join Assume $M > \sqrt{F|R|}$

$$
\begin{aligned}
N_S &= 2 + \lceil \frac{|R|}{O} \rceil + \lceil \frac{|S|}{O} \rceil + 2B && (28.21)\\
N_{I/O} &= \lceil \frac{|R|}{I_1} \rceil + \lceil \frac{|R|}{O} \rceil + \lceil \frac{|S|}{I_1} \rceil + \lceil \frac{|S|}{O} \rceil + B + \lceil \frac{|S|}{I_2} \rceil && (28.22)\\
N_X &= 3|R| + 3|S| && (28.23)\\
O &= \lfloor \frac{M - I_1}{B} \rfloor && (28.24)\\
B &= \lceil \frac{|R|F}{M - I_1} \rceil && (28.25)\\
& && (28.26)
\end{aligned}
$$

B is the number of buckets, I_1 is the input buffer size for the first phase, I_2 is the input buffer size of the second phase

Hybrid Hash Join Assume $M > \sqrt{F|R|}$

$$
\begin{aligned}
N_S &= 2 + \lceil \frac{|R'|}{O} \rceil + \lceil \frac{|S'|}{O} \rceil + 2K && (28.27)\\
N_X &= |R| + |S| + 2|R'| + 2|S'| && (28.28)\\
N_{I/O} &= \lceil \frac{|R|}{I_1} \rceil + \lceil \frac{|R'|}{O} \rceil + \lceil \frac{|S|}{I_1} \rceil + \lceil \frac{|S'|}{O} \rceil + K + \lceil \frac{|S'|}{I_2} \rceil && (28.29)\\
|R_0| &= \lfloor \frac{WS}{F} \rfloor && (28.30)\\
|R'| &= |R| - |R_0| && (28.31)\\
|S'| &= \lceil |S| * (1 - \frac{|R_0|}{|R|}) \rceil && (28.32)\\
K &= \lceil \frac{|R|F - (M - I_1)}{M - I_2 - O} \rceil && (28.33)\\
WS &= M - K * O - I_1 && (28.34)\\
& && (28.35)
\end{aligned}
$$

where K is the smallest K for which

$$K * (M - I_2) + WS > |R|F$$

28.4 Sorting, Grouping, and Duplicate Elimination

28.5 Bibliography

[920] [928]

[87] [98] [101] [157] [156] [197] [196] [225] [260] [381] [446] [447] [448] [450] [504] [495] [500] [522] [549] [568] [833] [832] [873] [576] [887] [889] [931] [932]

Hybrid hash join: [650]

Part VII

Implementation

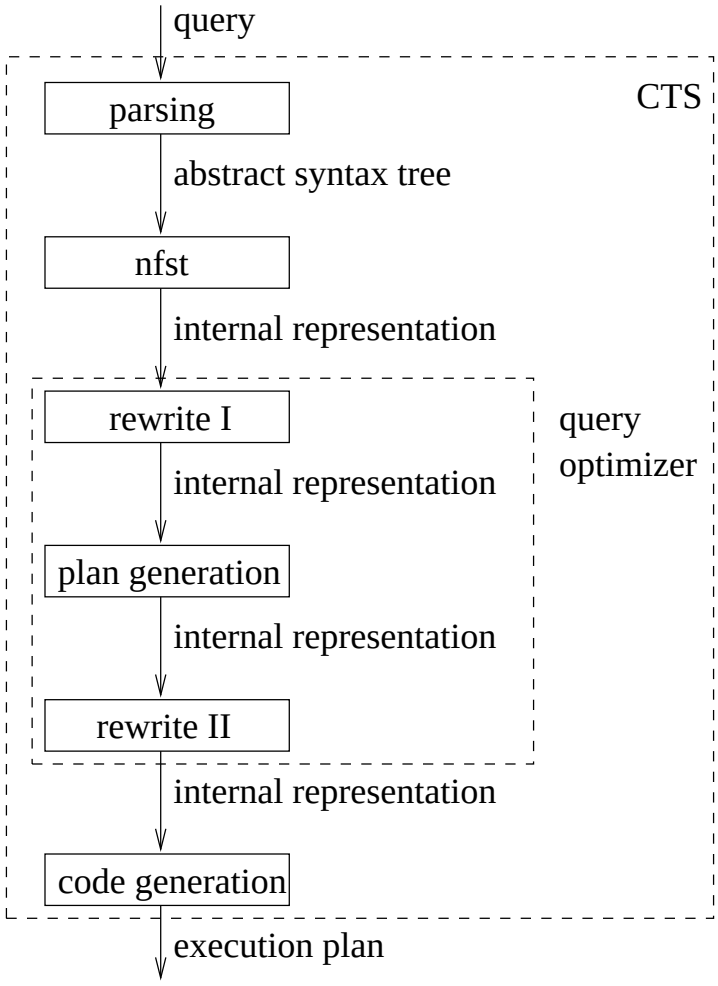


Figure 29.1: The compilation process

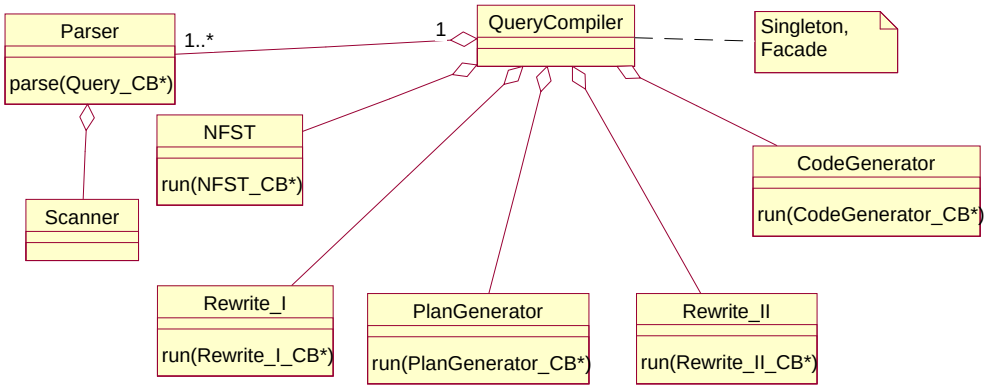


Figure 29.2: Class Architecture of the Query Compiler

Chapter 29

Architecture of a Query Compiler

29.1 Compilation process

29.2 Architecture

Figure 29.1 a path of a query through the optimizer. For every step, a single component is responsible. Providing a facade for the components results in the overall architecture (Fig. 29.2). Every component is reentrant and stateless. The information necessary for a component to process a query is passed via references to control blocks. Control blocks are discussed next, then we discuss memory management. Subsequent sections describe the components in some detail.

29.3 Control Blocks

It is very convenient to have a hierarchy of control blocks within the optimizer. Figure 29.3 shows some of the control blocks. For simplification, those blocks concerned with session handling and transaction handling are omitted. Every routine call within the optimizer has a control block pointer as a parameter. The routines belonging to a specific phase have a pointer to the phase' specific control block as a parameter.

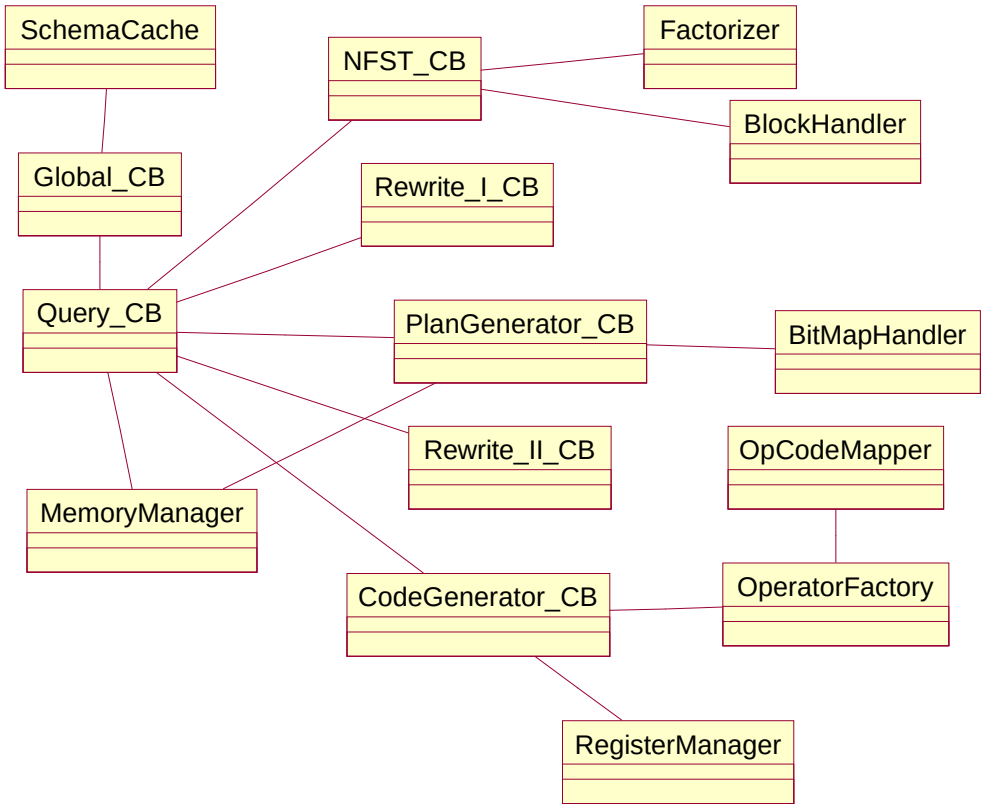


Figure 29.3: Control Block Structure

For example, the routines in NFST have a `NFST_CB` pointer as a parameter. We now discuss the purpose of the different control blocks.

The global control block governs the behavior of the query compiler. It contains boolean variables indicating which phases to perform and which phases of the compilation process are to be traced. It also contains indicators for the individual phases. For example, for the first rewrite phase it contains indicators which rules to apply, which rules to trace and so on. These control indicators are manipulated by the driver which also allows to step through the different phases. This is very important for debugging purposes. Besides this overall control of the query compilers behavior, the global control block also contains a pointer to the schema cache. The schema cache itself allows to look up type names, relations, extensions, indexes, and so on.

The query control block contains all the information gathered for the current query so far. It contains the abstract syntax tree, after its construction, the analyzed and translated query after NFST has been applied, the rewritten plan after the `Rewrite_I` phase, and so on. It also contains a link to the memory manager that manages memory for this specific query. After the control block for a query is created, the memory manager is initialized. During the destructor call, the memory manager is destroyed and memory released.

Some components need helpers. These are also associated with the control blocks. We discuss them together with the components.

29.4 Memory Management

There are three approaches to memory management in query optimizers. The first approach is to use an automatic garbage collector if the language provides one. This is not necessarily the most efficient approach but by far the most convenient one. This approach can be imitated by an implementation based on smart pointers. I would not recommend doing so since the treatment of cycles can be quite tricky and it is inefficient. Another approach would be to collect all references to newly created objects and release these after the query has been processed. This approach is easy to implement, very convenient (transparent to the implementor), but inefficient. A better approach is to allocate bigger areas of memory by a memory manager. Factories[1] then use these memory chunks to generate objects as necessary. After the query has been processed, the chunks are freed.

Here, we consider only memory whose duration lasts for the processing of a single query. In general, we have more kinds of memory whose validity conforms to sessions and transactions.

29.5 Tracing and Plan Visualization

29.6 Driver

29.7 Bibliography

[1] Design pattern.

Chapter 30

Internal Representations

30.1 Requirements

easy access to information

query representation: overall design goal: methods/functions with semantic meaning, not only syntactic meaning.

relationships: consumer/producer (occurrance) precedence order information equivalence of expressions (transitivity of equality) see also expr.h fuer andere funktionen/beziehungen die gebraucht werden

2-ebenen repraesentation. 2. ebene materialisiert einige beziehungen und funktionen, die haeufig gebraucht werden und kompliziert zu berechnen sind anderer grund fuer materialisierung: vermeide zuviele geschachtelte forschleifen. bsp: keycheck: gegeben eine menge von attributen und eine menge von schluesseln, ist die menge ein schluessel? teste jeden schluessel, kommt jedes element in schluessel in menge von attributen vor? (schon drei schleifen!!!)

modellierungsdetail: ein grosser struct mit dicken case oder feine klassenhierarchie. wann splitten: nur wenn innerhalb des optimierers verschiedene abarbeitung erfordert.

Representation: info captured: 1) 1st class information (information obvious in original query+(standard)semantic analysis) 2) 2nd class information (derived information) 3) historic information (during query optimization itself) - modified (original expression, modifier) - copied (original expression, copier) 4) information about the expression itselt: (e.g.: is_function_call, is_select) 5) specific representations for specific purposes (optimization algorithms, code generation, semantic analysis) beziehungen zwischen diesen repraesentationen

info captured for 1) different parts of the optimizer

syntactic/semantic information

garbage collection: 1) manually 2) automatic 3) semi-automatic (collect references, free at end of query)

30.2 Algebraic Representations

relational algebra in: [192].

30.2.1 Graph Representations

30.2.2 Query Graph

also called *object graph*: [74, 927]

30.2.3 Operator Graph

used in: [794], [923]
enhanced to represent physical properties: [707]
with outerjoins: [704], [293]
graph representation and equivalence to calculus: [635]

30.3 Query Graph Model (QGM)

30.4 Classification of Predicates

klassifikation von praedikaten

- nach stelligkeit,wertigkeit (selektion, join, nasty)
- nach funktor(=,¡,..., between, oder boolsche funktion)
- nach funktion: fuer keys in index: start/stop/range/exact/enum range(in-predicate)
- nach sel-wert: simple (col = const), komplex (col = expr) cheap/expensive
- nach join wert: fuer hj, smj, hbnlj,...
- korrelationspraedikate

30.5 Treatment of Distinct

30.6 Query Analysis and Materialization of Analysis Results

Questions:

1. was materialisieren wir
2. was packen wir in die 1. repraesentation?
   - bsp: properties: zeiger auf property oder besser inline properties
   - bsp: unique number: entweder in expr oder getrennter dictionary struktur

```
query analysis (purpose, determine optimization algorithm)
#input relations, #predicates, #ex-quantifiers, #all-quantifiers,
#conjunctions, #disjunctions, #joingraphkind(star,chain,tree,cyclic)
#strongly-connected-components (for crossproduct indication)
#false aggregates in projection list clause (implies grouping required)
/* remark: typical query optimizes should at least have two algorithms:
```

```
   - exhaustive  (for large queries)
   - heuristic   (for small queries)
*/
```

for blocks: indicator whether they should produce a null-tuple, in case they do not produce any tuple. this is nice for some rewrite rules. other possibility: if-statement in algebra.

30.7 Query and Plan Properties

Logical and Physical Properties of Plans

Ausführungsplänen können eine Reihe von Eigenschaften zugeordnet werden. Diese Eigenschaften fallen in drei Klassen

1. logische Eigenschaften, also beispielsweise
   (a) beinhaltete Relationen
   (b) beinhaltete Attribute
   (c) angewendete Prädikate
2. physische Eigenschaften, also beispielsweise
   (a) Ordnung der Tupel
   (b) Strom oder Materialisierung des Ergebnisses
   (c) Materialisierung im Hauptspeicher oder Hintergrundspeicher
   (d) Zugriffspfade auf das Ergebnis
   (e) Rechnerknoten des Ergebnis (im verteilten Fall)
   (f) Kompression
3. quantitative Eigenschaften, also beispielsweise
   (a) Anzahl der Elemente im Ergebnis
   (b) Größe des Ergebnisses oder eines Ergebniselementes
   (c) Auswertungskosten aufgeschlüsselt nach I/O, CPU und Kommunikationskosten

kosten: diese sind zu berechnen und dienen als grundlage fuer die planbewertung ges-kosten /* gesamt kosten (ressourcenverbrauch) */ ges-kosten += cpu-instr / inst/sek ges-kosten += seek-kosten * overhead (waiting/cpu) ges-kosten += i/o-kosten * io-weight cpu-kosten /* reine cpu-kosten */ i/o-kosten /* hintergrundspeicherzugriff (warten auf platte + cpu fuer seitenzugriffe) */ com-kosten /* kommunikation */ com-init /* initialisierungskosten fuer kommunikationsvorgang */ com-exit /* exitkosten fuer kommunikationsvorgang */ com-cptu /* kosten fuer jede transfereinheit (z.b. byte) waehrend eines kommunikationsvorgangs */

kostenstruktur koennte etwas sein, dass ges/cpu/io kosten enthaelt. ausserdem waeren kosten fuer rescanning interessant, falls dies notwendig ist (pufferprobleme,

indexscan und dann faellt seite raus) weiteres interessantes kostenmass sind die kosten, bis das erste tupel berechnet wird.

dies sind die konstanten, die system-abhaengig sind. am besten sind, sie werden gemessen. Hardware: #cpu-instruktionen pro sekunde #cpu-instruktionen fuer block zugriff/transfer lesen/schreiben #cpu-instruktionen pro transfer init/send/exit init/receive/exit ms fuer seek/latency/transfer pro nK block

RTS-kosten #cpu-instruktionen fuer open/next/close fuer scan operatoren unter verschiedenen voraussetzungen:mit/ohne praedikat, mit/ohne projektion (entsprechend den avm programmen) #cpu-instruktionen fuer open/next/close fuer jeden alg operator, #cpu-instruktionen fuer funktionen/operationen/praedikate/avm-befehle

statistics: first/large physical page of a relation number of pages of a relation -¿ to estimate scan cost measured sequential scan cost (no interference/plenty interference)

–properties:

- menge der quns
- menge der attribute
- menge der praedikate
- ordnung
- boolean properties
- globale menge der gepipelineten quns
- kostenvektor
- cardinalitaeten bewiesen/geschaetzt
- gewuenschter puffer
- schluessel, fds
- #seiten, die durch ein fetch gelesen werden sollen
- menge der objekte, von denen der plan (der ja teilplan sein kann) abhaengt
- eigenschaften fuer parallele plaene
- eigenschaften fuer smp plaene

```
das folgende ist alles blabla. aber es weisst auf den punkt hin,
das in dieser beziehung etwas getan werden muss.
--index: determine degree of clustering
  - lese_rate = #gelesene_seiten / seiten_fuer_relation
    ein praedikate erniedrigt die lesen_rate, ein erneutes lesen aufgrund
    falls TIDs sortiert werden, muss fetch_ration erneut berechnet werden
  - seiten koennen in gruppen z.b. auf einem zylinder zusammengefasst wer
    und mit einem prefetch befehl geholt werden. anzahl seeks abschaetzen
  - cluster_ration(CR)
    CR = P(read(t) ohne page read) = (card - anzahl pagefetch)/card
```

```
                                    = (card - (#pagefetch - #page))/card
    das ist besonderer  quark
  - cluster_factor(CF)
    CF = P(avoid unnecessary pagefetch) = (pagefetch/maxpagefetch)
       = card -#fetch / card - #pageinrel
    das ist besonderer  quark
index retrieval on full key => beide faktoren auf 100% setzen, da
  innerhalb eines index die TIDs pro key-eintrag sortiert werden.
```

Speicherung von Properties unter dynamischem Programmieren und Memoization: Kosten und andere Eigenschaften, die nicht vom Plan abhängen, können pro Planklasse gespeichert werden und brauchen nicht pro Plan gespeichert zu werden.

30.8 Conversion to the Internal Representation

30.8.1 Preprocessing

30.8.2 Translation into the Internal Representation

30.9 Bibliography

Chapter 31

Details on the Phases of Query Compilation

31.1 Parsing

Lexical analysis is pretty much the same as for traditional compilers. However, it is convenient to treat keywords as soft. This allows for example for relation names like *order* which is a keyword in SQL. This might be very convenient for users since SQL has plenty (several hundreds) of keywords. For some keywords like *select* there is less danger of it being a relation name. A solution for *group* and *order* would be to lex them as a single token together with the following *by*.

Parsing again is very similar to parsing in compiler construction. For both, lexing and parsing, generators can be used to generate these components. The parser specification of SQL is quite lengthy while the one for OQL is pretty compact. In both cases, a LALR(2) grammar suffices. The outcome of the parser should be an abstract syntax tree. Again the data structure for abstract syntax trees (ast) as well as operations to deal with them (allocation, deletion, traversal) can be generated from an according ast specification.

During parsing already some of the basic rewriting techniques can be applied. For example, `between` can be eliminated.

In BD II, there are currently four parsers (for SQL, OQL, NQL (a clean version of XQuery), XQuery). The driver allows to step through the query compiler and allows to influence its overall behavior. For example, several trace levels can be switched on and off while within the driver. Single rewrites can be enabled and disabled. Further, the driver allows to switch to a different query language. This is quite convenient for debugging purposes. We used the Cocktail tools to generate the lexer, parser, ast, and NFST component.

31.2 Semantic Analysis, Normalization, Factorization, Constant Folding, and Translation

The NFST component performs (at least) four different tasks:

1. normalization of expressions,

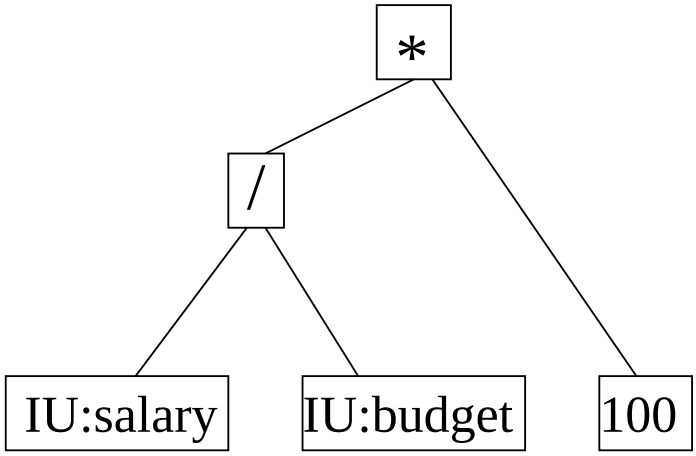


Figure 31.1: Expression

2. factorization of common subexpressions,

3. semantic analysis, and

4. translation into the internal algebra-based query representation.

Although these are different tasks, a single pass over the abstract syntax tree suffices to perform all these tasks in one step.

Consider the following example query:

select e.name, (d.salary / d.budget) * 100
from Employee e, Department d
where e.salary $>$ 100000 **and** e.dno = d.dno

The internal representation of the expression *(d.salary / d.budget) * 100* in the query is shown in Fig. 31.1. It contains two operator nodes for the operations "$*$" and "$/$". At the bottom, we find IU nodes. IU stands for Information Unit. A single IU corresponds to a variable that can be bound to a value. Sample IUs are attributes of a relation or, as we will see, intermediate results. In the query representation, there are three IUs. The first two IUs are bound to attribute values for the attributes *salary* and *budget*. The third IU is bound to the constant 100.

NFST routines can be implemented using a typical compiler generator tool. It is implemented in a rule-based language. Every rule matches a specific kind of AST nodes and performs an action. The ast tree is processed in post order.

The hierarchy for organizing different kinds of expressions is shown in Fig 31.2. Here is a list of useful functions:

- occurrance of expressions in another expression
- for a given expression: compute the set of occurring (consumed, free) IUs
- for a given expression: compute the set of produced IUs

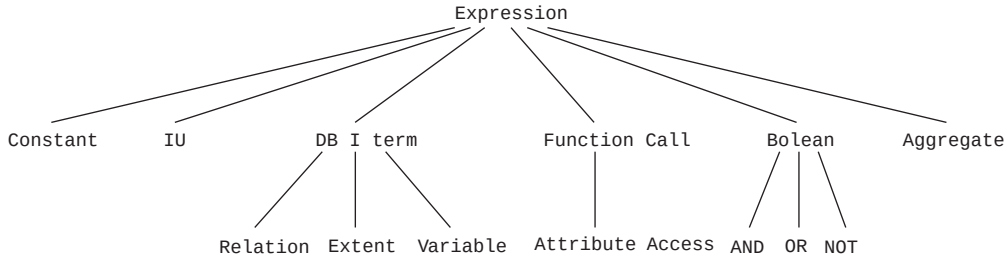


Figure 31.2: Expression hierarchy

- for a given IU, retrieve the block producing the IU
- determine whether some block returns a single value only
- computation of the transivitity of predicates, especially equality to derive its equivalence classes.
- determine whether some expression produces a subset of another expression
- constant folding
- merge and/or (from e.g. binary to n-ary) and push not operations
- replace a certain expression by another one
- deep and shallow copy

These functions can be implemented either as member functions of expressions or according to visitor/collector/mutator patterns. For more complex functions (consumer/producer) we recommend the latter.

Some of these functions will be called quite frequently, e.g. the consumer/producer, precedence ordering, equivalence (transivitity of equality) functions. So it might be convenient to compute these relationships only once and then materialize them. Since some transformation in the rewrite phases are quite complex, a recomputation of these materialized functions should be possible since their direct maintenance might be too complex.

31.3 Normalization

Fig. 31.3 shows the result after normalization. The idea of normalization is to introduce intermediate IUs such that all operators take only IUs as arguments. This representation is quite useful.

31.4 Factorization

Common subexpressions are factorized by replacing them with references to some IU. For the expressions in TPCD query 1, the result is shown in Fig. 31.4. Factorization is enabled by a factorization component that takes care of all expressions seen so far

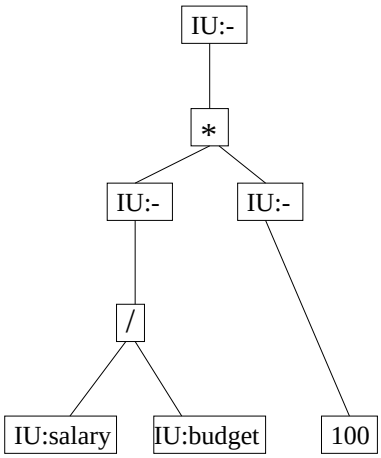


Figure 31.3: Expression

and the IUs representing these expressions. Every expression encountered by some NFST routine is passed to the factorization. The result is a reference to an IU. This IU can be a new IU in case of a new expression, or an existing IU in case of a common subexpression. The factorization component is available to the NFST routines via the NFST control block which is associated with a factorization component (Fig.29.3).

31.5 Constant Folding

31.6 Semantic analysis

The main purpose of semantic analysis is to attach a type to every expression. For simple expressions it is very similar to traditional semantic analysis in compiler construction. The only difference occurs for references to schema constructs. The schema is persistence and references to e.g. relations or named objects have to be looked up there. For performance reasons it is convenient to have a schema cache in order to cache frequently used references. Another aspect complicating semantic analysis a little is that collection types are frequently used in the database context. Their incorporation is rather straight forward but the different collection types should be handled with care.

As programming languages, query languages provide a block structure. Consider for example the SQL query

```
...
      select  a, b, c
      from    A, B
      where   d > e and f = g
...
```

Consider the semantic analysis of *d*. Since SQL provides *implicit name look up,* we have to check (formerly analyzed) relations A and B whether they provide an attribute called *d*. If none of them provides an attribute *d*, then we must check the next upper

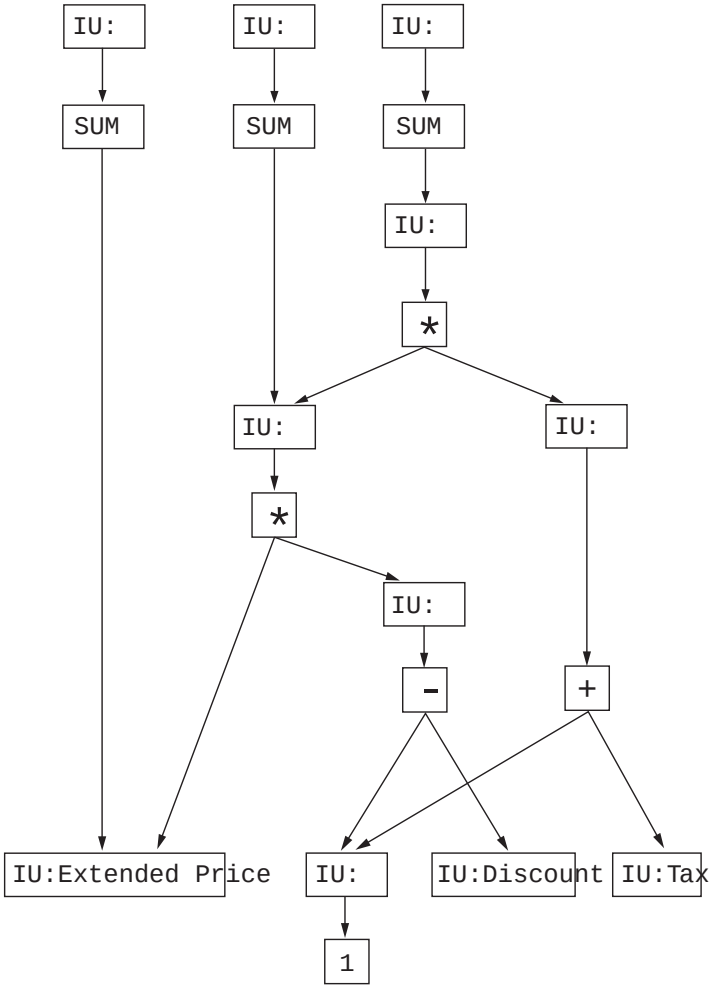


Figure 31.4: Query 1

SFW-block. If at least one of the relations A or B provides an attribute d, we just check that only one of them provides such an attribute. Otherwise, there would be an unallowed ambiguity. The blockwise look up is handled by block handler. For every newly encounterd block (e.g. SFW block), a new block is opened. All identifiers analyzed within that block are pushed into the list of identifiers for that block. In case the query language allows for implicit name resolution, it might also be convenient to push all the attributes of an analyzed relation into the blocks list. The lookup is then performed blockwise. Within every block, we have to check for ambiguities. If the lookup fails, we have to proceed looking up the identifier in the schema. The handling of blocks and lookups is performed by the BlockHandler component attached to the control block of the NFST component (Fig. 29.3).

Another departure from standard semantic analysis are *false aggregates* as provided by SQL.

select avg(age)
from Students

I call *count(age)* a *false aggregate* since a true aggregate function operators on a collection of values and returns a single value. Here, the situation is different. The attribute *age* is of type integer. Hence, for the average function whith signature *avg :* $\{int\} \longrightarrow int$ the semantic analysis would detect a typing error. The result is that we have to treat these false aggregates as special cases. This is (mostly) not necessary for query languages like OQL.

31.7 Translation

The translation step translates the original AST representation into an internal representation. There are as many internal query representations as there are query compiler. They all build on calculus expressions, operator graphs build over some algebra, or tableaux representations [848, 849]. A very powerful representation that also captures the subtleties of duplicate handling is the query graph model (QGM) [661].

The representation we use here is a mixture of a typed algebra and calculus. Algebraic expressions are simple operator trees with algebraic operators like selection, join, etc. as nodes. These operator trees must be correctly typed. For example, we are very picky about whether a selection operator returns a set or a bag. The expression that more resemble a calculus representation than an algebraic expression is the *SFWD block* used in the internal representation. We first clarify our notion of block within the query representation described here and then give an example of an SFWD block. A block is everything that produces variable bindings. For example a SFWD-block that pretty directly corresponds to a SFW-block in SQL or OQL. Other examples of blocks are quantifier expressions and grouping operators. A block has the following ingredients:

- a list of inputs of type collection of tuples[1] (labeled *from*)
- a set of expressions whose top is an IU (labeled *define*)
- a selection predicate of type bool (labeled *where*)

For *quantifier blocks* and *group blocks*, the list of inputs is restricted to length one. The SFWD-block and the grouping block additionally have a projection list (labeled *select*) that indicates which IUs are to be projected (i.e. passed to subsequent operators). Blocks are typed (algebraic) expressions and can thus be mixed with other expressions and algebraic operator trees.

An example of a SFWD-block is shown in Fig. 31.5 where dashed lines indicate the *produced-by* relationship. The graph corresponds to the internal representation of our example query. The semantics of a SFWD-block can be described as follows. First, take the cross product of the collections of tuples found in the list of inputs. (If this is not possible, due to dependencies, d-joins have to be used.) Then, for every resulting tuple, compute the bindings for all the IUs mentioned in the *define* clause, apply the selection predicate and return all the bindings for the IUs mentioned in the *select* clause.

Although the SFWD-block looks neat, it lacks certain information that must be represented. This information concerns the role of the entries in the *from* clause and

[1] We use a quite general notion of tuple: a tuple is a set of variable (IU) bindings.

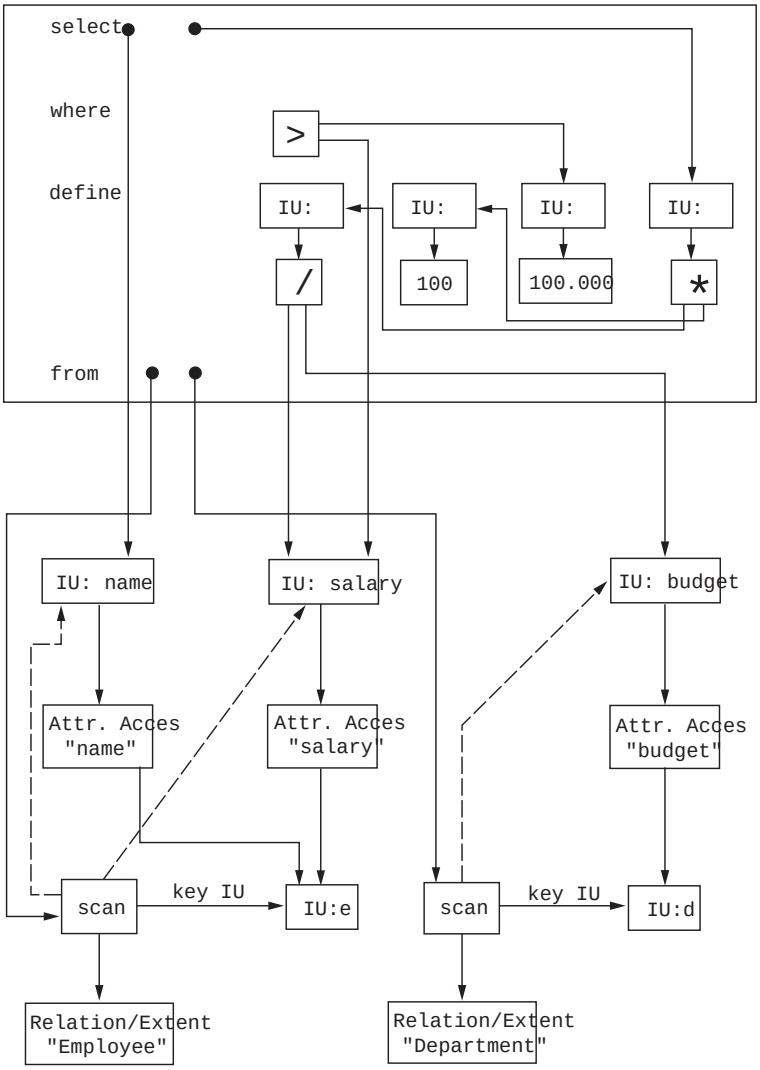


Figure 31.5: Internal representation

duplicate elimination. Let us start with the latter. There are three views relevant to duplicate processing:

1. the user view: did the user specify distinct?
2. the context view: does the occurrence or elimination of duplicates make a difference for the query result?
3. the processing view: does the block produce duplicates?

All this information is attached to a block. This information can then be summarized to one of three values representing

- eliminate duplicates
- preserve duplicates

- don't care about duplicates
  (The optimizer can feel free to do whatever is more efficient.)

This summary is also attached to every block. Let us illustrate this by a simple example:

select **distinct** ssno
from Employee
where ... **and**
exists(**select** ... **from** ... **where**)

For the inner block, the user specifies that duplicates are to be preserved. However, duplicates or not does not modify the outcome of *exists*. Hence, the contextual information indicates that the outcome for the inner block is a don't care. The processing view can determine whether the block produces duplicates. If for all the entries in the *from* clause, a key is projected in the *select* clause, then the query does not produce duplicates. Hence, no special care has to be taken to remove duplicates produced by the outer block if we assume that *ssno* is the key of *Employee*.

No let us consider the annotations for the arguments in the *from* clause. The query

select **distinct** e.name
from Employee e, Department d
where e.dno = d.dno

retrieves only *Employee* attributes. Such a query is most efficiently evaluated by a semi-join. Hence, we can add a semi-join (SJ) annotation to the *Department d* clause.

For queries without a **distinct**, the result may be wrong (e.g. in case an employee works in several departments) since a typical semi-join just checks for existence. A special semi-join that preserves duplicates should be used. The according annotation is (SJ,PD). Another annotation occurs whenever an outer-join is used. Outer joins can (in SQL) be part of the *from* clause. Typically they have to be fully parenthesized since outer joins and regular joins not always commute. But under special circumstances, they commute and hence a list of entries in the *from* clause suffices [292]. Then, the entry to be preserved (the outer part) should be annotated by (OJ). We use (AJ) as the anti-join annotation, and (DJ) for a d-join. To complete annotation, the case of a regular join can be annotated by (J). If the query language also supports all-quantifications, that translate to divisions, then the annotation (D) should be supported.

Since the graphical representation of a query is quite complex, we also use text representations of the result of the NFST phase. Consider the following OQL query:

select distinct s.name, s.age, s.supervisor.name, s.supervisor.age
from s in Student
where s.gpa $>$ 8 **and** s.supervisor.age $<$ 30

The annotated result (without duplicate annotations) of the normalization and factorization steps is

select distinct sn, sa, ssn, ssa
from s in Student (J)
where sg $>$ 8 **and** ssa$<$ 30
define sn = s.name
sg = s.gpa
sa = s.age
ss = s.supervisor
ssn= ss.name
ssa= ss.age

Semantic analysis just adds type information (which we never show).

In standard relational query processing multiple entries in the **from** clause are translated into a cross product. This is not always possible in object-oriented query processing. Consider the following query

select distinct s
from s **in** Student, c **in** s.courses
where c.name = “Database”

which after normalization yields

select distinct s
from s in Student, c **in** s.courses
where cn = “Database”
define cn = c.name

The evaluation of *c in s.courses* is dependend on *s* and cannot be evaluated if no *s* is given. Hence, a cross product would not make much sense. To deal with this situation, the *d-join* has been introduced [186]. It is a binary operator that evaluates for every input tuple from its left input its right input and flattens the result. Consider the algebraic expression given in Fig. 31.6. For every student *s* from its left input, the d-join computes the set *s.courses*. For every course c in *s.courses* an output tuple containing the original student s and a single course c is produced. If the evaluation of the right argument of the d-join is not dependend on the left argument, the d-join is equivalent with a cross product. The first optimization is to replace d-joins by cross products whenever possible.

Queries with a **group by** clause must be translated using the *unary grouping* operator GROUP which we denote by Γ. It is defined as

$$\begin{aligned}\Gamma_{g;\theta A;f}(e) &= \{y.A \circ [g : G] | y \in e, \\ &\quad G = f(\{x | x \in e, x.A\theta y.A\})\}\end{aligned}$$

where the subscripts have the following semantics: (i) g is a new attribute that will hold the elements of the group (ii) θA is the grouping criterion for a sequence of comparison operators θ and a sequence of attribute names A, and (iii) the function f will be applied to each group after it has been formed. We often use some abbreviations. If the

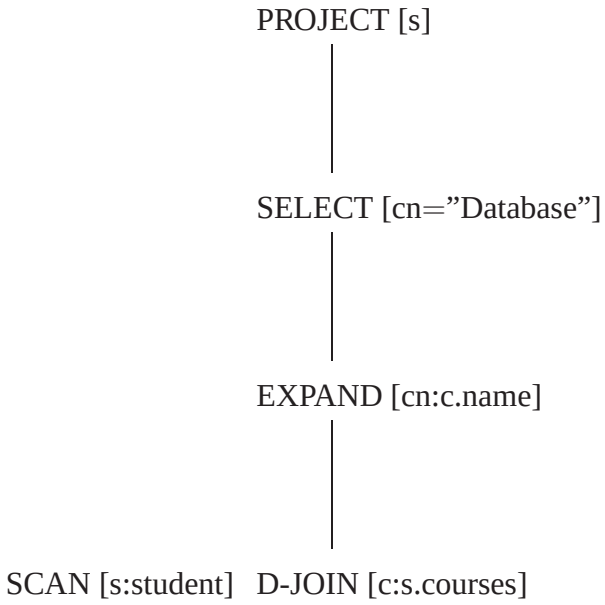


Figure 31.6: An algebraic operator tree with a d-join

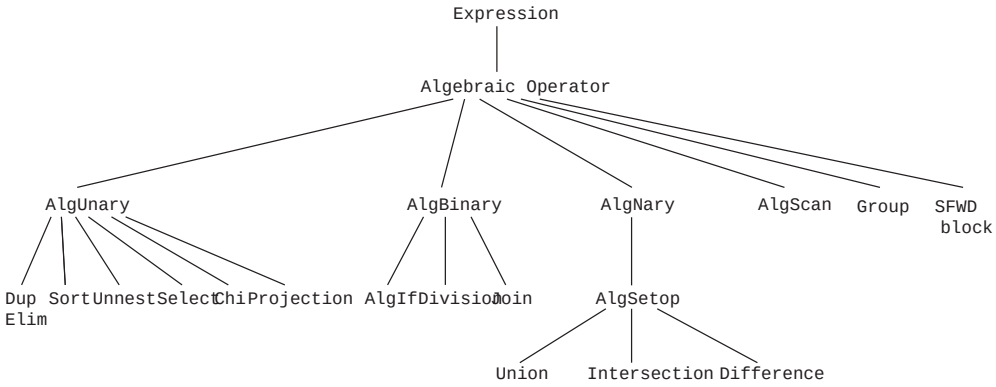


Figure 31.7: Algebra

comparison operator θ is equal to "=", we don't write it. If the function f is identity, we omit it. Hence, $\Gamma_{g;A}$ abbreviates $\Gamma_{g;=A;id}$.

Let us complete the discussion on internal query representation. We already mentioned algebraic operators like selection and join. These are called logical algebraic operators. There implementations are called physical algebraic operators. Typically, there exist several possible implementations for a single logical algebraic operator. The most prominent example being the join operator with implementations like Grace join, sort-merge join, nested-loop join etc. All the operators can be modelled as objects. To do so, we extend the expression hierarchy by an algebra hierarchy. Although not shown in Fig 31.7, the algebra class should be a subclass of the expression class. This is not necessary for SQL but is a requirement for more orthogonal query languages like OQL.

31.8 Rewrite I

31.9 Plan Generation

31.10 Rewrite II

31.11 Code generation

In order to discuss the tasks of code generation, it is necessary to have a little understanding of the interface to the runtime system that interpretes the execution plan. I have chosen AODB as an example runtime system since this is one I know. The interface to AODB is defined by the AODB Virtual Machine (AVM). For simple operations, like arithmetic operations, comparisons and so on, AVM provides assembler-like operations that are interpreted at runtime. Simple AVM operations work on *registers*. A single register is able to hold the contents of exactly one IU. Additionally, AVM provides physical algebraic operators. These operators take AVM programs (possibly with algebraic operators) as arguments. There is one specialty about AVM programs though. In order to efficiently support factorization of common subexpressions involving arithmetic operations (as needed in aggregations like avg, sum), arithmetic operators in AVM can have two side effects. They are able to store the result of the operation into a register and they are able to add the result of the operation to the contents of another register. This is denoted by the result mode. If the result mode is A, they just add the result to some register, if it is C, they copy (store) the result to some register, if it is B, they do both. This is explored in the code for Query 1 of the TPC-D benchmark (Fig. 1.6).

Code generation has the following tasks. First it must map the physical operators in a plan to the operators of the AVM code. This mapping is a straight forward 1:1 mapping. Then, the code for the subscripts of the operators has to be generated. Subscripts are for example the predicate expressions for the selection and join operators. For grouping, several AVM programs have to be generated. First program is the *init* program. It initializes the registers that will hold the results for the aggregate functions. For example, for an average operation, the register is initalized with 0. The *advance* program is executed once for every tuple to advance the aggregate computation. For example, for an average operations, the value of some register of the input tuple is added to the result register holding the average. The *finalize* program performs post-processing for aggregate functions. For example for the average, it devides the sum by the number of tuples. For hash-based grouping, the last two programs (see Fig.1.6) compute the hash value of the input register set and compare the group-by attributes of the input registers with those of every group in the hash bucket.

During the code generation for the subscripts factorization of common subexpression has to take place. Another task is register allocation and deallocation. This task is performed by the register manager. It uses subroutines to determine whether some registers are no longer needed. The register manager must also keep track in which register some IU is stored (if at all). Another component used during code generation is a factory that generates new AVM operations. This factory is associated with a table driven component that maps the operations used in the internal query representation to

AVM opcodes.

31.12 Bibliography

Chapter 32

Quality Assurance

32.1 Verification

32.2 Validation

32.3 Debugging

32.4 Test Data Generation

32.5 Benchmarking

Validation/Verification/Testing/Debugging/Benchmarking

- sql generation: [791, 805, 867, 868]
- verification: ftp/coco_plans.pdf
- debugging: ftp/vcoco.pdf
- test data generation:
- benchmarks:

32.6 Bibliography

Part VIII

Selected Topics

Chapter 33

Generating Plans for Top-N-Queries?

33.1 Motivation and Introduction

motivation:

- first by user (ordered)
- optimize for n rows (user/cursor)
- exist(subquery) optimize for 1 row
- having count(*) $<=$ n

33.2 Optimizing for the First Tuple

33.3 Optimizing for the First N Tuples

- nl-join instead of sm/hash join
- index access over table scan
- disable prefetching

[113, 114, 115] [136, 238] [258, 259, 423] [539]
[369] (also contains inverted list algorithms under frequent updates)
[539]

Chapter 34

Recursive Queries

Chapter 35

Issues Introduced by OQL

35.1 Type-Based Rewriting and Pointer Chasing Elimination

The first rewrite technique especially tailored for the object-oriented context is *type-based rewriting*. Consider the query

select distinct sn, ssn, ssa
from s **in** Student

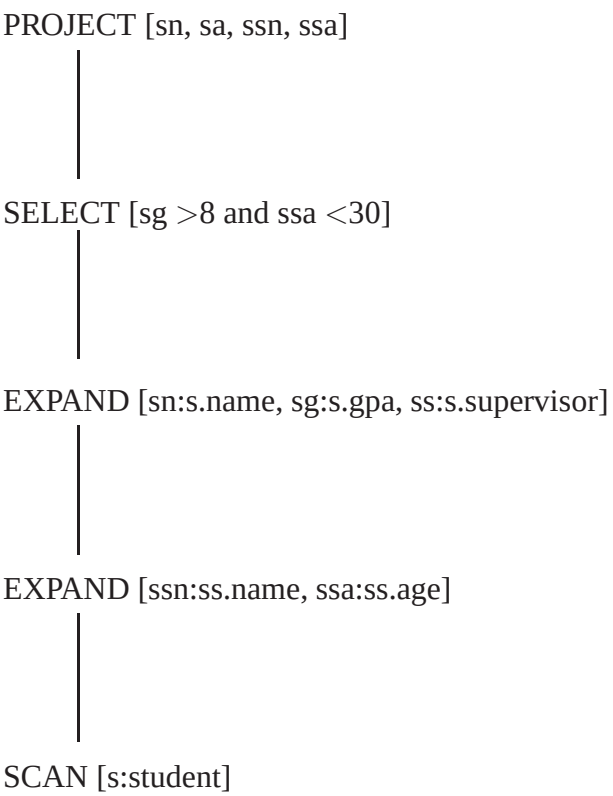


Figure 35.1: Algebraic representation of a query

```
where   sg > 8 and ssa < 30
define  sn = s.name
        sg = s.gpa
        ss = s.supervisor
        ssn= ss.name
        ssa= ss.age
```

The algebraic expression in Fig. 35.1 implies a scan of all students and a subsequent dereferentiation of the *supervisor* attribute in order to access the supervisors. If not all supervisors fit into main memory, this may result in many page accesses. Further, if there exists an index on the supervisor's *age*, and the selection condition *ssa < 30* is highly selective, the index should be applied in order to retrieve only those supervisors required for answering the query. Type-based rewriting enables this kind of optimization. For any expression of certain type with an associated extent, the extent is introduced in the *from* clause. For our query this results in

```
select distinct sn, pn, pa
from    s in Student, p in Professor
where   sg > 8 and pa < 30 and ss = p
define  sn = s.name
        sg = s.gpa
        ss = s.supervisor
        pn = ss.name
        pa = ss.age
```

As a side-effect, the attribute traversal from students via supervisor to professor is replaced by a join. Now, join-ordering allows for several new plans that could not be investigated otherwise. For example, we could exploit the above mentioned index to retrieve the young professors and join them with the students having a *gpa* greater than 8. The according plan is given in Fig. 35.2. Turning implicit joins or pointer chasing into explicit joins which can be freely reordered is an original query optimization technique for object-oriented queries. Note that the plan generation component is still allowed to turn the explicit join into an implicit join again.

Consider the query

```
select distinct p
from    p in Professor
where   p.room.number = 209
```

Straight forward evaluation of this query would scan all professors. For every professor, the *room* relationship would be traversed to find the room where the professor resides. Last, the room's number would be retrieved and tested to be 209. Using the *inverse relationship*, the query could as well be rewritten to

```
select distinct r.occupiedBy
from    r in Room
```

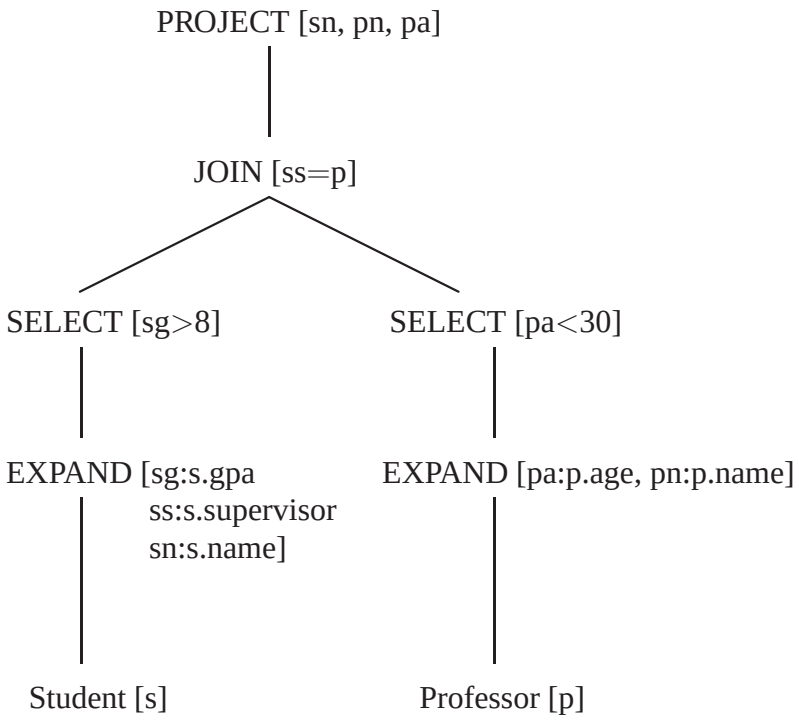


Figure 35.2: A join replacing pointer chasing

where r.number = 209

The evaluation of this query can be much more efficient, especially if there exists an index on the room number. Rewriting queries by exploiting inverse relationships is another rewrite technique to be applied during Rewrite Phase I.

35.2 Class Hierarchies

Another set of equivalences known from the relational context involves the UNION operator ($\cup$) and plays a vital role in dealing with class/extent hierarchies. Consider the simple class hierarchy given in Figure 35.3. Obviously, for the user, it must appear that the extent of *Employee* contains all *Manager*s. However, the system has different alternatives to implement extents. Most OBMSs organize an object base into areas or volumes. Each area or volume is then further organized into several files. A file is a logical grouping of objects not necessarily consisting of subsequent physical pages on disk. Files don't share pages.

The simplest possible implementation to scan all objects belonging to a certain extent is to perform an area scan and select those objects belonging to the extent in question. Obviously, this is far to expensive. Therefore, some more sophisticated possibilities to realize extents and scans over them are needed. The different possible implementations can be classified along two dimensions. The first dimension distinguishes between logical and physical extents, the second distinguishes between strict and (non-strict) extents.

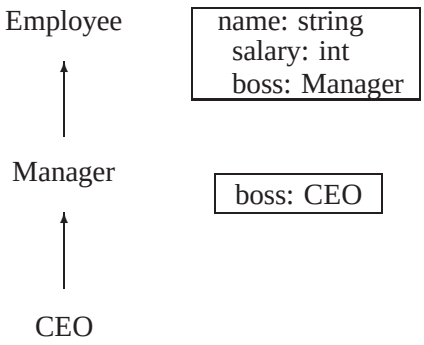


Figure 35.3: A Sample Class Hierarchy

Logical *vs.* Physical Extents

An extent can be realized as a collection of object identifiers. A scan over the extent is then implemented by a scan over all the object identifiers contained in the collection. Subsequently, the object identifiers are dereferenced to yield the objects themselves. This approach leads to logical extents. Another possibility is to implement extent membership by physical containment. The best alternative is to store all objects of an extent in a file. This results in physical extents. A scan over a physical extent is then implemented by a file scan.

Extents *vs.* Strict Extents

A strict extent contains the objects (or their OIDs) of a class excluding those of its subclasses. A non-strict extent contains the objects of a class and all objects of its subclasses.

Given a class C, any strict extent of a subclass C' of C is called a subextent of C.

Obviously, the two classifications are orthogonal. Applying them both results in the four possibilities presented graphically in Fig. 35.4. [191] strongly argues that strict extents are the method of choice. The reason is that only this way the query optimizer might exploit differences for extents. For example, there might be an index on the *age* of *Manager* but not for *Employee*. This difference can only be exploited for a query including a restriction on *age*, if we have strict extents.

However, strict extents result in initial query plans including UNION operators. Consider the query

select e
from e **in** Employee
where e.salary $>$ 100.000

The initial plan is

$$\sigma_{sa>100.000}(\chi_{sa:x.salary}((Employee[x] \cup Manager[x]) \cup CEO[x]))$$

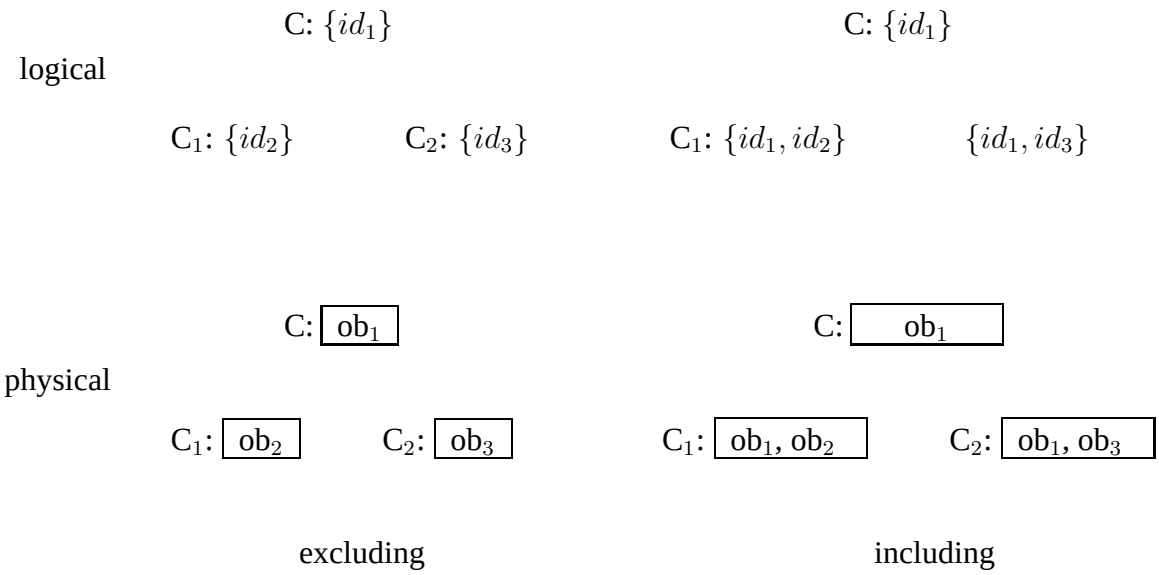


Figure 35.4: Implementation of Extents

Hence, algebraic equivalences are needed to reorder UNION operators with other algebraic operators. The most important equivalences are

$$e_1 \cup e_2 \equiv e_2 \cup e_1 \tag{35.1}$$
$$e_1 \cup (e_2 \cup e_3) \equiv (e_1 \cup e_2) \cup e_3 \tag{35.2}$$
$$\sigma_p(e_1 \cup e_2) \equiv \sigma_p(e_1) \cup \sigma_p(e_2) \tag{35.3}$$
$$\chi_{a:e}(e_1 \cup e_2) \equiv \chi_{a:e}(e_1) \cup \chi_{a:e}(e_2) \tag{35.4}$$
$$(e_1 \cup e_2) \bowtie_p e_3 \equiv (e_1 \bowtie_p e_3) \cup (e_2 \bowtie_p e_3) \tag{35.5}$$

Equivalences containing the UNION operator sometimes involve tricky typing constraints. These go beyond the current chapter and the reader is refered to [597].

35.3 Cardinalities and Cost Functions

Chapter 36

Issues Introduced by XPath

36.1 A Naive XPath-Interpreter and its Problems

36.2 Dynamic Programming and Memoization

[329, 331, 330]

36.3 Naive Translation of XPath to Algebra

36.4 Pushing Duplicate Elimination

36.5 Avoiding Duplicate Work

36.6 Avoiding Duplicate Generation

[407]

36.7 Index Usage and Materialized Views

[46]

36.8 Cardinalities and Costs

36.9 Bibliography

Chapter 37

Issues Introduced by XQuery

37.1 Reordering in Ordered Context

37.2 Result Construction

[271, 272] [765]

37.3 Unnesting Nested XQueries

Unnesting with error: [646]
[578, 580, 579, 581]

37.4 Cardinalities and Cost Functions

cardinality: [156, 905, 906, 729] [7]
XPathLearner: [545]
Polyzotis et al (XSKETCH): [667, 668, 665], [669]

37.5 Bibliography

[582] [847] [226]
Numbering: [270] Timber [445] TAX Algebra [449], physical algebra of Timber [647]
Structural Joins [24, 796]
SAL: [66], TAX: [449], XAL: [278]

- XML Statistics for hidden web: [10]
- XPath selectivity for internet scale: [7]
- StatiX: [279]
- IMAX: incremental statistics [683]
- Metrics for XML Document Collections: [487]

- output size containment join: [874]
- Bloom Histogram: [875]

View and XML: [2]
Quilt: [125]
Timber: [445] Monet: [741] Natix: NoK: [934]
Correlated XPath: [935]
Wood: [897, 898, 899]
Path based approach to Storage (XRel): [926]
Grust: [361, 363, 362, 364, 840]
Liefke: Loop fusion etc.: [544]
Benchmarking: XMach-1: [89], MBench: [719] XBench: [640, 918, 919], XMark: [742] XOO7: [97]
Rewriting: [230, 353, 354]
[229, 418]
Incremental Schema Validation: [93, 645]
Franklin (filtering): [233]

Chapter 38

Outlook

What we did not talk about: multiple query optimization, semantic query optimization, special techniques for optimization in OBMSs, multi-media data bases, object-relational databases, spatial databases, temporal databases, and query optimization for parallel and distributed database systems.

Multi Query Optimization? [753]

Parametric/Dynamic/Adaptive Query Optimization? [33, 34, 35, 31, 351, 344] [441, 442, 463, 852]
[43]

Parallel Database Systems?

Distributed Database Systems? [498]

Recursive Queries?

Multi Database Systems?

Temporal Database Systems?

Spatial Database Systems?

Translation of Triggers and Updates?

Online Queries (Streams)?

Approximate Answers? [321]

Appendix A

Query Languages?

A.1 Designing a query language

requirements
    design principles for object-oriented query languages: [417] [79]

A.2 SQL

A.3 OQL

A.4 XPath

A.5 XQuery

A.6 Datalog

Appendix B

Query Execution Engine (?)

- Overview Books: [397, 300]
- Overview: Graefe [338, 339]
- Implementation of Division [335, 343, 345]
- Implementation of Division and set-containment joins [679]
- Hash vs. Sort: [340, 348]
- Heap-Filter Merge Join: [337]
- Hash-Teams

Appendix C

Glossary of Rewrite and Optimization Techniques

trivopt Triviale Auswertungen bspw. solche für widersprüchliche Prädikate werden sofort vorgenommen. Dies ist eine Optimierungstechnik, die oft bereits auf der Quellebene durchgeführt wird.

pareval Falls ein Glied einer Konjunktion zu *false* evaluiert, werden die restlichen Glieder nicht mehr evaluiert. Dies ergibt sich automatisch durch die Verwendung von hintereinanderausgeführten Selektionen.

pushnot Falls ein Prädikat die Form $\neg(p_1 \wedge p_2)$ hat, so ist *pareval* nicht anwendbar. Daher werden Negationen nach innen gezogen. Auf $\neg p_1 \vee \neg p_2$ ist *pareval* dann wieder anwendbar. Das Durchschieben von Negationen ist auch im Kontext von NULL-Werten unabdingbar für die Korrektheit. Dies ist eine Optimierungstechnik, die oft bereits auf der Quellebene durchgeführt wird.

bxp Verallgemeinert man die in *pareval* und *notpush* angesprochene Problematik, so führt dies auf die Optimierung von allgemeinen booleschen Prädikaten.

trans Durch Ausnutzen der Transitivität von Vergleichsoperationen können neue Selektionsprädikate gewonnen und Konstanten propagiert werden. Diese Optimierungstechnik erweitert den Suchraum und wird ebenfalls auf der Quellebene durchgeführt. Bei manchen Systemen wir dieser Schritt nicht durchgeführt, falls sehr viele Relationen zu joinen sind, um den Suchraum nicht noch weiter zu vergrößern [313, 314].

selpush Selektionen werden so früh wie möglich durchgeführt. Diese Technik führt nicht immer zu optimalen Auswertungsplänen und stellt somit eine Heuristik dar. Diese Optimierungstechnik schränkt den Suchraum ein.

projpush Die Technik zur Behandlung von Projektionen ist nicht ganz so einfach wie die der Selektion. Zu unterscheiden ist hier, ob es sich um eine Projektion mit Duplikateliminierung handelt oder nicht. Je nach dem ist es sinnvoll, die Projektion zur Wurzel des Operatorgraphen zu verschieben oder zu den Blättern hin. Die Projektion verringert den Speicherbedarf von Zwischenergebnissen,

da die Tupel weniger Attribute enthalten. Handelt es sich um eine duplikateliminierende Projektion, so wird möglicherweise auch die Anzahl der Tupel verringert. Duplikatelimination als solche ist aber eine sehr teure Operation. Diese wird üblicherweise durch Sortieren implementiert. Bei großen Datenmengen gibt es allerdings bessere Alternativen. Auch Hash-basierte Verfahren eignen sich zur Duplikateliminierung. Diese Optimierungstechnik schränkt den Suchraum ein.

grouppush Pushing a grouping operation past a join can lead to better plans.

crossjoin Ein Kreuzprodukt, das von einer Selektion gefolgt wird, wird wenn immer möglich in eine Verbundoperation umgewandelt. Diese Optimierungstechnik schränkt den Suchraum ein, da Pläne mit Kreuzprodukten vermieden werden.

nocross Kreuzprodukte werden wenn immer möglich vermieden oder, wenn dies nicht möglich ist, erst so spät wie möglich durchgeführt. Diese Technik verringert den Suchraum, führt aber nicht immer zu optimalen Auswertungsplänen.

semjoin Eine Verbundoperation kann durch eine Semiverbundoperation ersetzt werden, wenn nur die Attribute einer Relation weitere Verwendung finden.

joinor Die Auswertungsreihenfolge von Verbundoperationen ist kritisch. Daher wurden eine Reihe von Verfahren entwickelt, die optimale oder quasi-optimale Reihenfolge von Verbundoperationen zu bestimmen. Oft wird dabei der Suchraum auf Listen von Verbundoperationen beschränkt. Die Motivation hierbei ist das Verkleinern des Suchraums und die Beschränkung auf nur eine zu erzeugenden Zwischenrelation. Dieses Verfahren garantiert nicht mehr ein optimales Ergebnis.

joinpush Tables that are guaranteed to produce a single tuple are always pushed to be joined first. This reduces the search space. The single tuple condition can be evaluated by determining whether all key attributes of a relation are fully qualified. [313, 314].

elimredjoin Eliminate redundant join operations. See Sections. . . XXX

indnest Eine direkte Evaluierung von geschachtelten Anfragen wird durch geschachtelte Schleifen vorgenommen. Dabei wird eine Unteranfrage für jede erzielte Bindung der äußeren Anfrage evaluiert. Dies erfordert quadratischen Aufwand und ist deshalb sehr ineffizient. Falls die innere Anfrage unabhängig von der äußeren Anfrage evaluiert werden kann, so wird diese herausgezogen und getrennt evaluiert. Weitere Optimierungen geschachtelter Anfragen sind möglich.

unnest Entschachtelung von Anfragen [186, 187, 298, 483, 488, 489, 661, 799, 801, 802]

compop Oft ist es sinnvoll, mehrere Operationen zu einer komplexeren zusammenzufassen. Beispielsweise können zwei hintereinander ausgeführte Selektionen durch eine Selektion mit einem komplexeren Prädikat ersetzt werden. Ebenso kann auch das Zusammenfassen von Verbundoperationen, Selektionen und Projektionen sinnvoll sein.

comsubexpr Gemeinsame Teilausdrücke werden nur einfach evaluiert. Hierunter fallen zum einen Techniken, die das mehrmalige Lesen vom Hintergrundspeicher verhindern, und zum anderen Techniken, die Zwischenergebnisse von Teilausdrücken materialisieren. Letzteres sollte nur dann angewendet werden, falls die k-malige Auswertung teurer ist als das einmalige Auswerten und das Erzeugen des Ergebnisses mit k-maligem Lesen, wobei k die Anzahl der Vorkommen im Plan ist.

dynminmax Dynamisch gewonnene Minima und Maxima von Attributwerten können für die Erzeugung von zusätzlichen Restriktionen herangezogen werden. Diese Technik funktioniert auch sehr gut für unkorrelierte Anfragen. Dabei werden min- und max-Werte herangezogen um zusätzliche Restriktionen für die Anfrage zu gewinnen. [488, 313, 314]

pma Predicate Move around moves predicates between queries and subqueries. Mostly they are duplicated in order to yield as many restrictions in a block as possible [536]. As a special case, predicates will be pushed into view definitions if they have to be materialized temporarily [313, 314].

exproj For subqueries with exist prune unnecessary entries in the **select** clause. The intention behind is that attributes projected unnecessarily might influence the optimizer's decision on the optimal access path [313, 314].

vm View merging expands the view definition within the query such that is can be optimized together with the query. Thereby, duplicate accesses to the view are resolved by different copies of the views definition in order to facilitate unnesting [313, 314, 661].

inConstSet2Or A predicate of the form $x \in \{a_1, \ldots, a_n\}$ is transformed into a sequence of disjunctions $x = a_1 \vee \ldots \vee x = a_n$ if the a_i are constants in order to allow index or-ing (TID list operations or bitvector operations) [313, 314].

like1 If the like predicate does not start with %, then a prefix index can be used.

like2 The pattern is analyzed to see whether a range of values can be extracted such that the pattern does not have to be evaluated on all tuples. The result is either a pretest or an index access. [313, 314].

like3 Special indexes supporting like predicates are introduced.

sort Vorhandene Sortierungen können für verschiedene Operatoren ausgenutzt werden. Falls keine Sortierung vorhanden ist, kann es sinnvoll sein, diese zu erzeugen [788]. Z.B. aufeinanderfolgende joins, joins und gruppierungen. Dabei kann man die Gruppierungsattribute permutieren, um sie mit einer gegebenen Sortierreihenfolge in Einklang zu bringen [313, 314].

aps Zugriffspfade werden eingesetzt, wann immer dies gewinnbringend möglich ist. Beispielsweise kann die Anfrage

select count(*) **from** R;

durch einen Indexscan effizient ausgewertet werden [160].

tmpidx Manchmal kann es sinnvoll sein, temporäre Zugriffspfade anzulegen.

optimpl Für algebraische Operatoren existieren im allgemeinen mehrere Implementierungen. Es sollte hier immer die für einen Operator im vorliegenden Fall billigste Lösung ausgewählt werden. Ebenfalls von Bedeutung ist die Darstellung des Zwischenergebnisses. Beispielsweise können Relationen explizit oder implizit dargestellt werden, wobei letztere Darstellung nur Zeiger auf Tupel oder Surrogate der Tupel enthält. Weitergedacht führt diese Technik zu den TID-Listen-basierten Operatoren.

setpipe Die Evaluation eines algebraischen Ausdrucks kann entweder mengenorientiert oder nebenläufig (pipelining) erfolgen. Letzteres erspart das Erzeugen von großen Zwischenergebnissen.

tmplay Das temporäre Ändern eines Layouts eines Objektes kann durchaus sinnvoll sein, wenn die Kosten, die durch diese Änderung entstehen, durch den Gewinn der mehrmaligen Verwendung dieses Layouts mehr als kompensiert werden. Ein typisches Beispiel ist *Pointer-swizzling*.

matSubQ If a query is not unnested, then for every argument combination passed to the subquery, the result is materialized in order to avoid duplicate computation of the same subquery expression for the same argument combination [313, 314]. This technique is favorable for detachment [816, 895, 927]

AggrJoin Joins with non-equi join predicates based on $\leq$ or $<$, can be processed more efficiently than by a cross product with a subsequent selection [189].

ClassHier Class hierarchies involve the computation of queries over a union of extents (if implemented that way). Pushing algebraic operations past unions allows often for more efficient plans [191].

AggrIDX Use an index to determine aggregate values like min/max/avg/count.

rid/tidsort When several tuples qualify during an index scan, the resulting TIDs can be sorted in order to guarantee sequential access to the base relation.

multIDX Perform operations like union and disjunction on the outcome of an index scan.

multIDXsplit If two ranges are queried within the same query ([1-10],[20-30]) consider multIDX or use a single scan through the index [1-30] with an additional qualification predicate.

multIDXor Queries with more conditions on indexed attributes can be evaluated by more complex combinations of index scans and tid-list/bitvector operations. (A = 5 and (B = 3 or B = 4)).

scanDirChange During multiple sequential scans of relation (e.g. for a block-wise nested loop join), the direction of the scan can be changed in order to reuse as much of the pages in the buffer as possible.

lock The optimizer should chose the correct locks to set on tables. For example, if a whole table is scanned, a table lock should be set.

expFunMat Expensive functions can be cached during query evaluation in order to avoid their multiple evaluation for the same arguments [405].

expFunFil Easiser to evaluate predicates that are implied by more expensive predicates can serve as filters in order to avoid the evaluation of the expensive predicate on all tuples.

stop Stop evaluation after the first tuple qualifies. This is good for existential subqueries, universal subqueries (disqualify), semi-joins for distinct results and the like.

expensive projections 1. zum Schluss, da dort am wenigsten verschiedene Werte

2. durchschieben, falls cache fuer Funktionsergebnisse dadurch vermieden werden kann

OO-Kontext: problematisch: objekte muessen fuer funktionen/methoden als ganzes vorhanden sein. daher ist eine einfache strategie nicht moeglich.

distinct/sorting

```
select distinct a,b,c
...
order by a,b
```

kann auch nach a,b,c sortiert werden. stoert gar nicht, vereinfacht aber die duplikateliminierung. nur ein sortieren notwendig.

index access

- by key
- by key range
- by dashed key range (set of keys/key ranges)
- index anding/oring

alternative operator implementations e.g. join: nlj bnlj hj grace-hash hybrid-hash smj diag-join star-join

distpd Push-down or Pull-up distinct.

aggregate with distinct

```
select a, agg(distinct b)
...
group by a
===>
sort on a,b
dup elim
group a,sum(b)
```

alternative: aggr(distinct *) is implemented such that it uses a hashtable to eliminate duplicates this is only good, if the number of groups is smalland the number of distinct values in each group is small.

XXX - use keys, inclusion dependencies, fds etc. (all user specified and derived) (propagate keys over joins as fds), (for a function call: derived IU is functional dependend on arguments of the function call if function is deterministic) (keys can be represented as sets of IUs or as bitvectors(given numbering of IUs)) (numbering inprecise: bitvectors can be used as filters (like for signatures))

Appendix D

Example Query Compiler

D.1 Research Prototypes

D.1.1 AQUA and COLA

D.1.2 Black Dahlia II

D.1.3 Epoq

Für das objektorientierte Datenmodell *Encore* [933] wurde die Anfragesprache Equal [772, 771, 773], eine objektorientierte Algebra, die die Erzeugung von Objekten erlaubt, entwickelt. Zur Optimierung von Equal-Algebra-Ausdrücken soll der Optimierer Epoq dienen. Eine Realisierung von Epoq steht noch aus. Konkretisiert wurden jedoch bereits der Architekturansatz [596] und die Kontrolle der Alternativenerzeugung [594] innerhalb dieser Architektur. Einen Gesamtüberblick gibt die Dissertation von Mitchell [593].

Der Architekturvorschlag besteht aus einer generischen Architektur, die an einem Beispieloptimierer konkretisiert wurde [593, 594]. Die elementaren Bausteine der Architektur sind Regionen. Sie bestehen aus einer Kontrollkomponente und wiederum Regionen beziehungsweise Transformationen. Die einfachste Region ist dabei eine Transformation/Regel, die einen Algebraausdruck in einen äquivalenten Algebraausdruck umformt. Jede Region selbst wird wiederum als eine Transformation aufgefaßt. Innerhalb der Architektur werden nun diese Regionen in einer Hierarchie oder auch einem gerichteten azyklischen Graphen, organisiert. Abbildung D.1 zeigt eine solche Beispielorganisation. Regionen selbst können bis auf die Kontrolle als Module im Sinne von Sciore und Sieg [746] aufgefaßt werden. Sie weisen sehr ähnliche Parameter und Schnittstellen auf. Während jedoch bei Sciore und Sieg die Kontrollstrategie eines Moduls aus einer festen Menge von gegebenen Kontrollstrategien ausgewählt werden muß, kann sie hier freier spezifiziert werden.

Unabhängig davon, ob die Transformationen einer Region wiederum Regionen sind oder elementare Transformationen, wird ihre Anwendung einheitlich von der Kontrolle der Region bestimmt. Die Aufgabe dieser Kontrolle besteht darin, eine Folge von Transformationen zu finden, die die gegebene Anfrage in eine äquivalente überführen. Sinngebend ist hierbei ein gewisses Ziel, das es zu erreichen gilt. Beispielsweise kann dieses Ziel lauten: Optimiere eine geschachtelte Anfrage. Um dieses

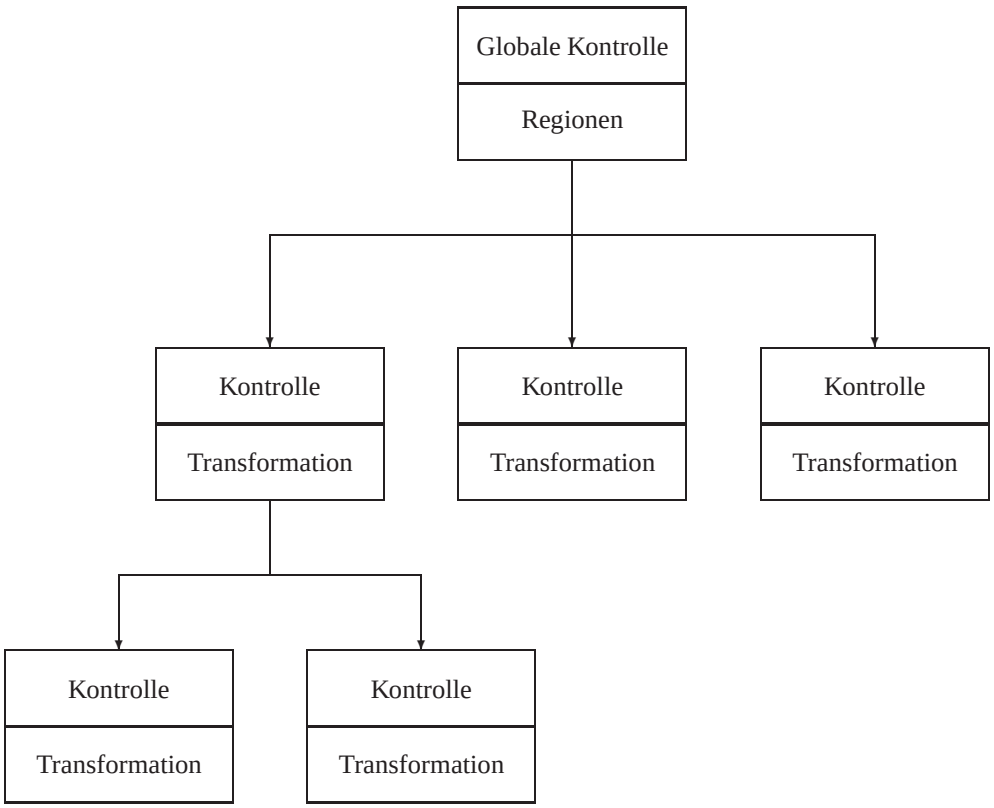


Figure D.1: Beispiel einer Epoq-Architektur

Ziel zu erreichen, sind zwei grobe Schritte notwendig. Zunächst muß die Anfrage entschachtelt werden und als nächstes die entschachtelte Anfrage optimiert werden. Man sieht sofort, daß die Folge der Transformationen, die die Kontrolle auszuwählen hat, sowohl von den Eigenschaften der Anfrage selbst wie auch vom zu erfüllenden Ziel abhängt. Basierend auf dieser Beobachtung wird die Kontrolle nicht als Suchfunktion implementiert, sondern es wird das Planungsparadigma zur Realisierung gewählt. Die Kontrolle selbst wird mit Hilfe eines Satzes von Regeln spezifiziert, die aus Vorbedingung und Aktion bestehen.

Da es nicht möglich ist, im Vorfeld einen Plan, also eine Sequenz von Transformationen/Regionen, zu erstellen, der in garantierter Weise das Ziel erreicht, wird erlaubt, daß die Ausführung einer Transformation/Region fehlschlägt. In diesem Fall kann dann ein alternativer Plan erzeugt werden, der aber auf dem bisher Erreichten aufsetzt. Hierzu werden die Regeln, die die Kontrolle spezifizieren in Gruppen eingeteilt, wobei jeder Gruppe eine einheitliche Vorbedingung zugeordnet ist. Zu jeder Gruppe gehört dann eine Sequenz von Aktionen, die der Reihe nach ausprobiert werden. Schlägt eine vorangehende Aktion fehl, so wird die nächste in der Reihe der Aktionen angewendet. Schlagen alle Aktionen fehl, so schlägt auch die Anwendung der Region fehl.

Jede Aktion selbst ist wiederum eine Sequenz von elementaren Aktionen. Jede dieser elementaren Aktionen ist entweder die Anwendung einer elementaren Transformation, der Aufruf einer Region oder der rekursive Aufruf des Planers mit einem neuformulierten Ziel, dessen Teilplan dann an entsprechender Stelle in die Aktion eingebaut wird.

Die Erweiterbarkeit dieses Ansatzes um neue Regionen scheint einfach möglich, da die Schnittstelle der Regionen genormt ist. Probleme könnte es lediglich bei den Kontrollstrategien geben, da nicht klar ist, ob die benutzte Regelsprache mächtig genug ist, um alle wünschenswerten Kontrollstrategien zu verwirklichen.

Die Frage, ob die einzelnen Komponenten des Optimierers, also die Regionen, evaluiert werden können, ist schwierig zu beantworten. Dafür spricht jedoch, daß jede Region in einem gewissen Kontext aufgerufen wird, also zur Erreichung eines bestimmten Zieles bei der Optimierung einer Anfrage mit ebenso bestimmten Eigenschaften. Beurteilen kann man daher die Erfolgsquote einer Region innerhalb ihrer verschiedenen Anwendungen. Da jede Region lediglich eine Alternative erzeugen darf, aufgrund des *eine Region ist eine Transformation*-Paradigmas, ist schwer zu sagen, in wieweit sich die durch die beschriebene Bewertung gewonnene Information zur Verbesserung der Regionen oder des Gesamtoptimierers einsetzen läßt.

Da auch hier der transformierende Ansatz zugrunde liegt, treffen die bereits diskutierten Probleme auch für den Optimierer für Straube zu.

Einen stetigen Leistungsabfall könnte man durch die Realisierung von alternativen Regionen erreichen, indem man ein Ziel *OptimiereSchnell* einführt, das dann entsprechend weniger sorgfältige, aber schnellere Regionen aufruft. Vorhersagen über der Güte (bei gegebener Optimierungszeit) scheinen aber schwerlich möglich.

D.1.4 Ereq

A primary goal of the EREQ project is to define a common architecture for the next generation of database managers. This architecture now includes

* the query language OQL (a la ODMG), * the logical algebra AQUA (a la Brown), and * the physical algebra OPA (a la OGI/PSU).

It also includes

* software to parse OQL into AQUA (a la Bolo)

and query optimizers:

* OPT++ (Wisconsin), * EPOQ (Brown), * Cascades (PSU/OGI), and * Reflective Optimizer (OGI).

In order to test this architecture, we hope to conduct a ”bakeoff” in which the four query optimizers will participate. The primary goal of this bakeoff is to determine whether optimizers written in different contexts can accommodate the architecture we have defined. Secondarily, we hope to collect enough performance statistics to draw some conclusions about the four optimizers, which have been written using significantly different paradigms.

At present, OGI and PSU are testing their optimizers on the bakeoff queries. Here is the prototype bakeoff optimizer developed at OGI. This set of Web pages is meant to report on the current progress of their effort, and to define the bakeoff rules. Please email your suggestions for improvement to Leo Fegaras fegaras@cse.ogi.edu. Leo will route comments to the appropriate author.

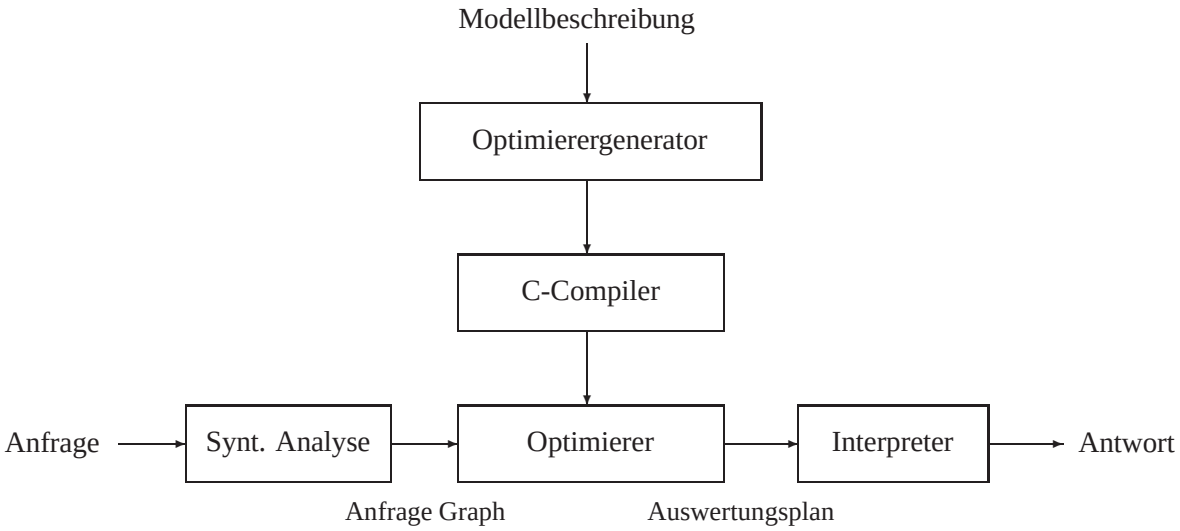


Figure D.2: Exodus Optimierer Generator

http://www.cse.ogi.edu/DISC/projects/ereq/bakeoff/bakeoff.html

D.1.5 Exodus/Volcano/Cascade

Im Rahmen des Exodus-Projektes wurde ein Optimierergenerator entwickelt [347]. Einen Überblick über den Exodus-Optimierergenerator gibt Abbildung D.2. Ein Model description file enthält alle Angaben, die für einen Optimierer nötig sind. Da der Exodus-Optimierergenerator verschiedene Datenmodelle unterstützen soll, enthält dieses File zunächst einmal die Definition der verfügbaren *Operatoren* und *Methoden*. Dabei werden mit *Operatoren* die Operatoren der logischen Algebra bezeichnet und mit *Methoden* diejenigen der physischen Algebra, also die Implementierungen der Operatoren. Das Model description file enthält weiterhin zwei Klassen von Regeln. *Transformationen* basieren auf algebraischen Gleichungen und führen einen Operatorbaum in einen anderen über. *Implementierungsregeln* wählen für einen gegebenen Operator eine Methode aus. Beide Klassen von Regeln haben einen linken Teil, der mit einem Teil des aktuellen Operatorgraphen übereinstimmen muß, einen rechten Teil, der den Operatorgraphen nach Anwendung der Regel beschreibt, und eine Bedingung, die erfüllt sein muß, damit die Regel angewendet werden kann. Während die linke und rechte Seite der Regel als Muster angegeben werden, wird die Bedingung durch C-Code beschrieben. Auch für die Tranformation lassen sich C-Routinen verwenden. In einer abschließenden Sektion des Model description files finden sich dann die benötigten C-Routinen.

Aus dem Model description file wird durch den Optimierergenerator ein C-Programm erzeugt, das anschließend übersetzt und gebunden wird. Das Ergebnis ist dann der Anfrageoptimierer, der in der herkömmlichen Art und Weise verwendet werden kann. Es wurde ein übersetzender Ansatz für die Regeln gewählt und kein interpretierender, da in einem von den Autoren vorher durchgeführten Experiment sich die Regelinterpretation als zu langsam erwiesen hat.

Die Regelabarbeitung im generierten Optimierer verwaltet eine Liste OPEN, in der alle anwendbaren Regeln gehalten werden. Ein Auswahlmechanismus bestimmt dann die nächste anzuwendende Regel und entfernt sie aus OPEN. Nach deren Anwendung werden die hierdurch ermöglichten Regelanwendungen detektiert und in OPEN vermerkt. Zur Implementierung des Auswahlmechanismus werden sowohl die Kosten eines aktuellen Ausdrucks als auch eine Abschätzung des Potentials einer Regel in Betracht gezogen. Diese Abschätzung des Potentials berechnet sich aus dem Quotienten der Kosten für einen Operatorbaum vor und nach Regelanwendung für eine Reihe von vorher durchgeführten Regelanwendungen. Mit Hilfe dieser beiden Angaben, den Kosten des aktuellen Operatorgraphen, auf den die Regel angewendet werden soll, und ihres Potentials können dann Abschätzungen über die Kosten des erzeugten Operatorgraphen berechnet werden. Die Suchstrategie ist Hill climbing.

Der von den Autoren vermerkte Hauptnachteil ihres Optimierergenerators, den sie jedoch für alle transformierenden regelbasierten Optimerer geltend machen, ist die Unmöglichkeit der Abschätzung der absoluten Güte eines Operatorbaumes und des Potentials eines Operatorbaumes im Hinblick auf zukünftige Optimierungen. Dadurch kann niemals abgeschätzt werden, ob der optimale Operatorbaum bereits erreicht wurde. Erst nach Generierung aller Alternativen ist die Auswahl des optimalen Operatorbaumes möglich. Weiter bedauern es die Autoren, daß es nicht möglich ist, den A$*$-Algorithmus als Suchfunktion zu verwenden, da die Abschätzung des Potentials oder der Distanz zum optimalen Operatorgraphen nicht möglich ist.

Zumindest kritisch gegenüberstehen sollte man auch der Bewertung einzelner Regeln, da diese, basierend auf algebraischen Gleichungen, von zu feiner Granularität sind, als daß eine allgemeine Bewertung möglich wäre. Die erfolgreiche Verwendung des Vertauschens zweier Verbundoperationen in einer Anfrage bedeutet noch lange nicht, daß diese Vertauschung auch in der nächsten Anfrage die Kosten verringert. Die Hauptursache für die kritische Einstellung gegenüber dieser recht ansprechenden Idee ist, daß eine Regelanwendung zu wenig Information/Kontext berücksichtigt. Würde dieses Manko beseitigt, wären Regeln also von entschieden gröberer Granularität, so erschiene dieser Ansatz vielversprechend. Ein Beispiel wäre eine Regel, die alle Verbundoperationen gemäß einer gegebenen Heuristik ordnet, also ein komplexer Algorithmus, der mehr Wissen in seine Entscheidungen einbezieht.

Graefe selbst führt einige weitere Nachteile des Exodus-Optimierergenerators an, die dann zur Entwicklung des Volcano-Optimierergenerators führten [349, 350]. Unzureichend unterstützt werden

- nicht-triviale Kostenmodelle,
- Eigenschaften,
- Heuristiken und
- Transformationen von Subskripten von algebraischen Operatoren in algebraische Operatoren.

Der letzte Punkt ist insbesondere im Bereich der Objektbanken wesentlich, um beispielsweise Pfadausdrücke in eine Folge von Verbundoperationen umwandeln zu können.

Im Volcano-Optimierergenerator werden algebraische Ausdrücke wieder in einen Operatorbaum umgewandelt. Wie im Exodus-Optimierergenerator wird der Optimierer wieder mit einer Menge von transformierenden und implementierenden Regeln beschrieben. Die Nachteile des transformierenden Ansatz werden somit geerbt. Eine Trennung in zwei Phasen, wie bei vielen Optimierern anzutreffen, ist für den Volcano-Optimierergenerator nicht notwendig. Der Entwickler des Optimierers hat die Freiheit, die Phasen selbst festzulegen. Die Probleme, die sonst bei der Kopplung der algebraischen mit der nicht-algebraischen Optimierung auftreten, können also vermieden werden. Die Behandlung der Eigenschaften erfolgt zielorientiert. Die in der Anfrage geforderten Eigenschaften (bspw. Sortierung), werden der Suchfunktion als Parameter übergeben, damit gezielt Pläne erstellt werden, die diese erfüllen. Wenn ein Operator oder eine Methode eingebaut wird, so wird darauf geachtet, daß diese noch nicht erfüllten Eigenschaften durch den Operator oder die Methode erzielt werden. Die geforderten Eigenschaften dienen wieder als Zielbeschreibung für die nachfolgenden Aufrufe der Suchfunktion. Zu diesen Eigenschaften gehören auch Kostengrenzen, mit denen die Suchfunktion dann einen Branch-and-bound-Algorithmus implementiert. Bevor ein Plan für einen algebraischen Ausdruck generiert wird, wird in einer Hash-Tabelle nachgeschaut, ob ein entsprechender Ausdruck mit den geforderten Eigenschaften bereits existiert. Dadurch wird Doppeltarbeit vermieden. Bei beiden Optimierergeneratoren werden die Forderungen nach stetigem Leistungsabfall, früher Bewertung von Alternativen und Evaluierbarkeit einzelner Komponenten nicht erfüllt.

D.1.6 Freytags regelbasierte System R-Emulation

[280] zeigt, wie man mit Hilfe eines regelbasierten Ansatzes den Optimierer von System R [750] emulieren kann. Die Eingabe besteht aus einem Lisp-ähnlichen Ausdruck:

(select <proj-list>
<sel-pred-list>
<join-pred-list>
<table-list>)

Die Projektionsliste besteht aus Attributspezifikationen der Form

<rel-name>.<attr-name>

Diese werden auch für die Selektionsprädikate und Joinprädikate verwendet. Die Algebra beinhaltet sowohl Operatoren der logischen als auch der physischen Algebra. Im einzelnen gibt es Scan-, Sort-, Projektions, Verbundoperatoren in einer logischen und verschiedenen physischen Ausprägungen. Die Erzeugung der Auswertungspläne wird in verschiedene Schritte unterteilt, die wiederum in Teilschritte zerlegt sind (siehe Abb. D.3). Zunächst erfolgt die Übersetztung in die logische Algebra. Hier werden Scan-Operatoren um die Relationen gebaut und Selektionen, die nur eine Relation betreffen, in die Scan-Opertoren eingebaut. Der zweite Schritt generiert Zugriffspläne, indem der Scan-Operator durch einen einfachen File-Scan (FSCAN) ersetzt wird, oder falls möglich, durch einen Index-Scan (ISCAN). Der dritte Schritt generiert zunächst verschiedene Verbund-Reihenfolgen und bestimmt anschließend die Verbund-Methoden. Sie in System R wird zwischen Sort-merge- und Nested-loop-join unterschieden.

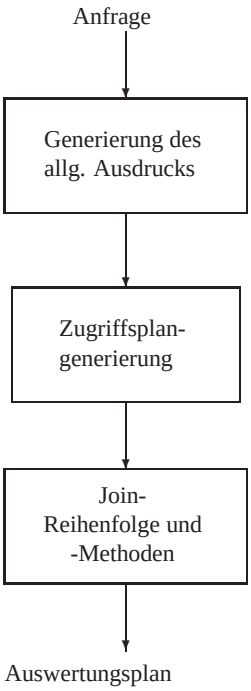


Figure D.3: Organisation der Optimierung

Es werden keinerlei Aussagen über die Auswahl einer Suchstrategie gemacht. Ziel ist es vielmehr, durch die Modellierung des System R Optimierers mit Hilfe eines Regelsystems die prinzipielle Brauchbarkeit des regelbasierten Ansatzes nachzuweisen.

D.1.7 Genesis

Das globale Ziel des Genesisprojektes [55, 56, 57, 60] war es, die gesamte Datenbanksoftware zu modularisieren und eine erhöhte Wiederverwendbarkeit von Datenbankmodulen zu erreichen. Zwei Teilziele wurden hierbei angestrebt:

1. Standardisierung der Schnittstellen und

2. Formulierung der Algorithmen unabhängig von der DBMS-Implementierung.

Wir interessieren uns hier lediglich für die Erreichung der Ziele beim Bau von Optimierern [53, 58].

Die Standardisierung der Schnittstellen wird durch eine Verallgemeinerung von Anfragegraphen erreicht. Die Algorithmen selbst werden durch Transformationen auf Anfragegraphen beschrieben. Man beachte, daß dies nicht bedeutet, daß die Algorithmen auch durch Transformationregeln implementiert werden. Regeln werden lediglich als Beschreibungsmittel benutzt, um die Natur der Wiederverwendbarkeit von Optimierungsalgorithmen zu verstehen.

Die Optimierung wird in zwei Phasen eingeteilt, die Reduktionsphase und die Verbundphase. Die Reduktionsphase bildet Anfragegraphen, die auf nicht reduzierten

Datenmengen arbeiten, auf solche ab, die auf reduzierten Datenmengen arbeiten. Die Reduktionsphase orientiert sich also deutlich an den Heuristiken zum Durchschieben von Selektionen und Projektionen. Die zweite Phase bestimmt Verbundordnungen. Damit ist die in den Papieren beschriebene Ausprägung des Ansatzes sehr konservativ in dem Sinne, daß nur klassische Datenmodelle betrachtet werden. Eine Anwendung der Methodik auf objektorientierte oder deduktive Datenmodelle steht noch aus.

Folglich lassen sich nur die existierenden klassischen Optimierungsansätze mit diesen Mitteln hinreichend gut beschreiben. Ebenso lassen sich die existierenden klassischen Optimierer mit den vorgestellten Mitteln als Zusammensetzung der ebenfalls im Formalismus erfaßten Algorithmen beschreiben. Die Zusammensetzung selbst wird mit algebraischen Termersetzungen beschrieben. Durch neue Kompositionsregeln lassen sich dann auch neue Optimierer beschreiben, die andere Kombinationen von Algorithmen verwenden.

Durch die formale, implementierungsunabhängige Beschreibung sowohl der einzelnen Optimierungsalgorithmen als auch der Zusammensetzung eines Optimierers wird die Wiederverwendbarkeit von bestehenden Algorithmen optimal unterstützt. Wichtig dabei ist auch die Verwendung der standardisierten Anfragegraphen. Dieser Punkt wird allerdings aufgeweicht, da auch vorgesehen ist, verschiedene Darstellungen von Anfragegraphen zu verwenden [56]. Hierdurch wird die Wiederverwendung von Implementierungen von Optimierungsalgorithmen natürlich in Frage gestellt, da diese üblicherweise nur auf einer bestimmten Darstellung der Anfragegraphen arbeiten.

Wenn neue Optimierungsansätze entwickelt werden, so lassen sie sich ebenfalls im vorgestellten Formalismus beschreiben. Gleiches gilt auch für neue Indexstrukturen, da auch diese formal beschrieben werden [54, 59]. Nicht abzusehen ist, in wieweit der standardisierte Anfragegraph Erweiterungen standhält. Dies ist jedoch kein spezifisches Problem des Genesisansatzes, sondern gilt für alle Optimierer. Es ist noch offen, ob es gelingt, die Optimierungsalgorithmen so zu spezifizieren und zu implementieren, daß sie unabhängig von der konkreten Darstellung oder Implementierung der Anfragegraphen arbeiten. Der objektorientierte Ansatz kann hier nützlich sein. Es erhebt sich jedoch die Frage, ob bei Einführung eines neuen Operators die bestehenden Algorithmen so implementierbar sind, daß sie diesen ignorieren können und trotzdem sinnvolle Arbeit leisten.

Die Beschränkung auf zwei Optimierungsphasen, die Reduktions- und die Verbundphase, ist keine Einschränkung, da auch sie mittels Termersetzungsregeln festgelegt wurde, und somit leicht geändert werden kann.

Da die Beschreibungen des Optimierers und der einzelnen Algorithmen unabhängig von der tatsächlichen Implementierung sind, sind auch die globale Kontrolle des Optimierers und die lokalen Kontrollen der einzelnen Algorithmen voneinander losgelöst. Dieses ist eine wichtige Forderung, um Erweiterbarkeit zu erreichen. Sie wird oft bei regelbasierten Optimierern verletzt und schränkt somit deren Erweiterbarkeit ein.

Die Evaluierbarkeit, die Vorhersagbarkeit und die frühe Bewertung von Alternativen sind mit dem vorgestellten Ansatz nicht möglich, da die einzelnen Algorithmen als Transformationen auf dem Anfragegraphen aufgefaßt werden. Dieser Nachteil gilt jedoch nicht allein für den hier vorgestellten Genesisansatz, sondern generell für alle bis auf einen Optimierer. Es ist allerdings nicht absehbar, ob dieser Nachteil aus dem verwendeten Formalismus resultiert oder lediglich aus deren Konkretisierung bei der Modellierung bestehender Optimierer. Es ist durchaus möglich, daß der Formalis-

mus mit leichten Erweiterungen auch andere Ansätze, insbesondere den generierenden, beschreiben kann.

Insgesamt handelt es sich beim Genesisansatz um einen sehr brauchbaren Ansatz. Leider hat er, im Gegensatz zur Regelbasierung, nicht genug Widerhall gefunden hat. Er hat höchst wahrscheinlich mehr Möglichkeiten, die Anforderungen zu erfüllen, als bisher ausgelotet wurde.

D.1.8 GOMbgo

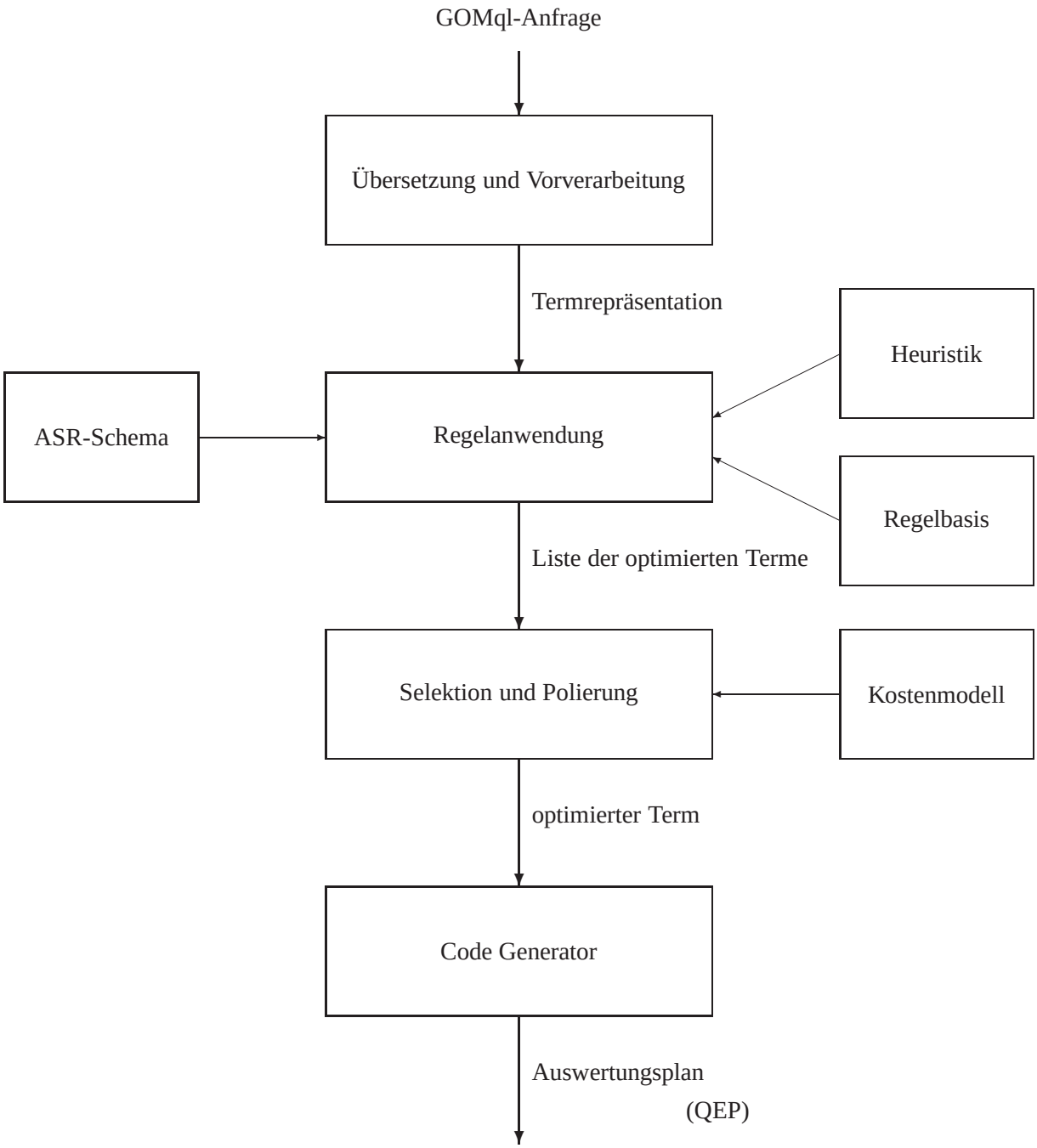


Figure D.4: Ablauf der Optimierung

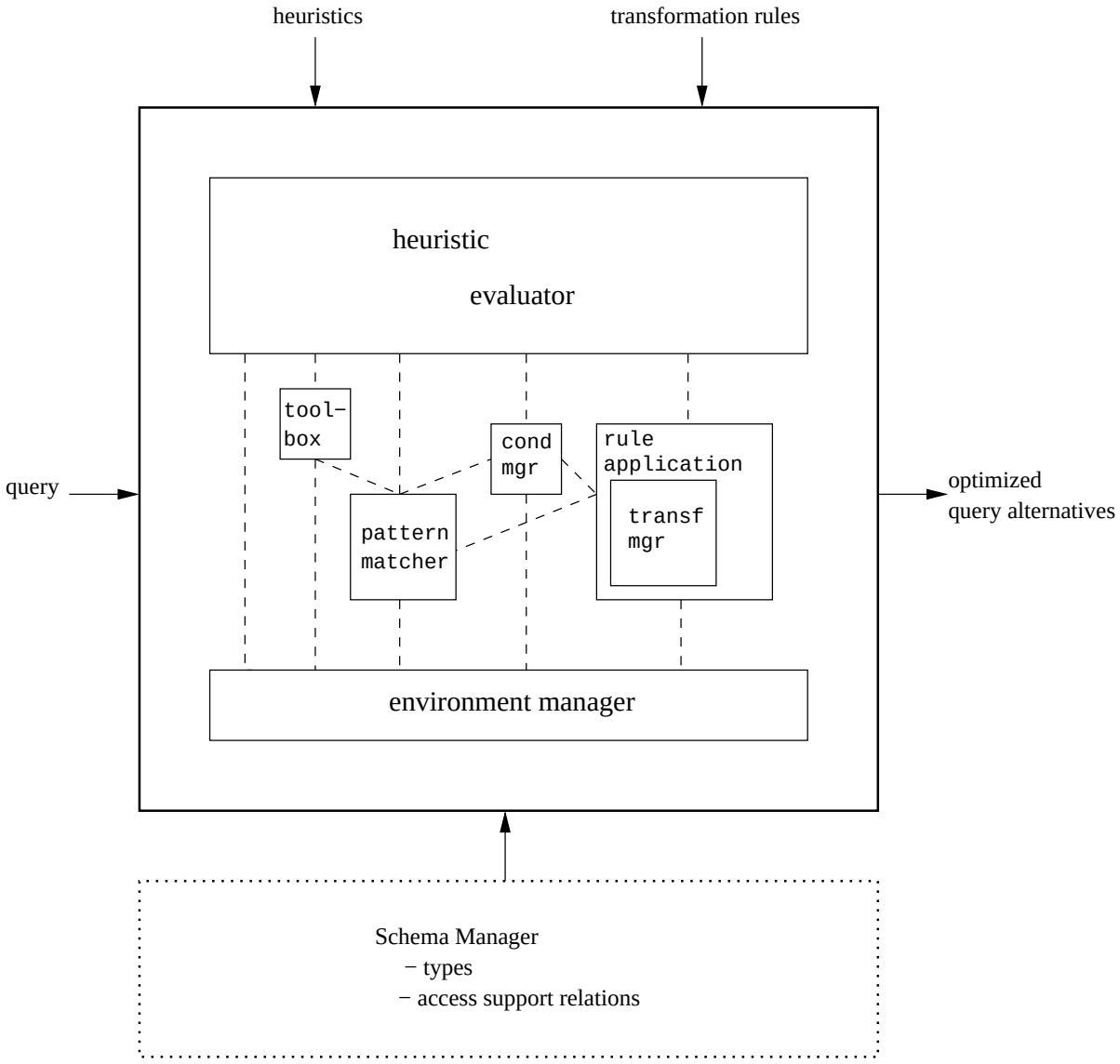


Figure D.5: Architektur von GOMrbo

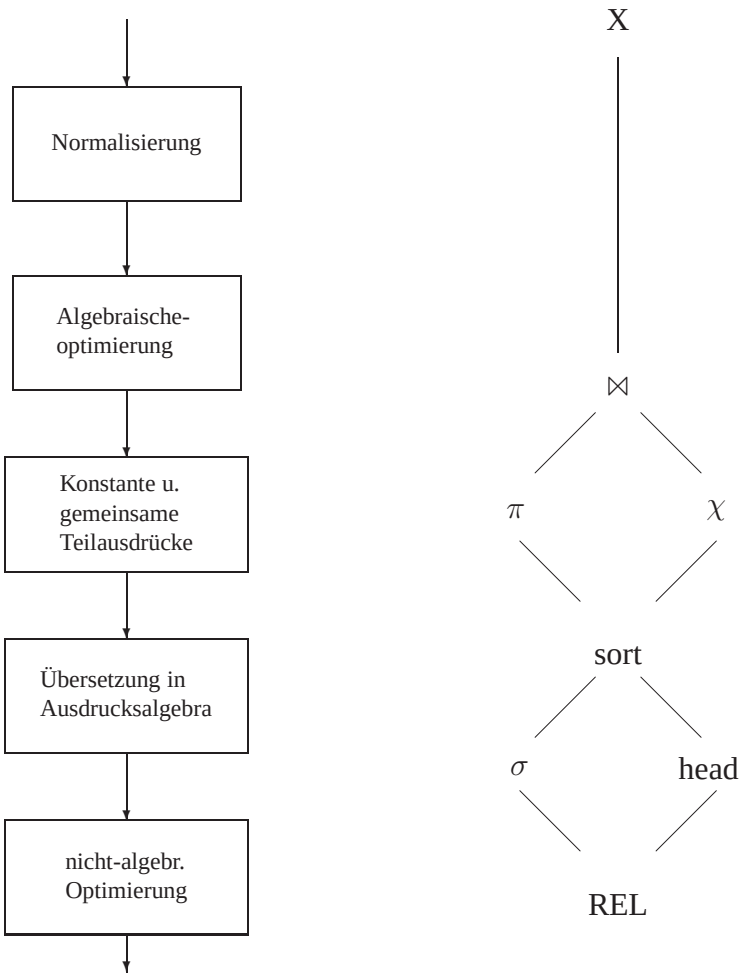


Figure D.6: a) Architektur des Gral-Optimierers; b) Operatorhierarchie nach Kosten

D.1.9 Gral

Gral ist ein erweiterbares geometrisches Datenbanksystem. Der für dieses System entwickelte Optimierer, ein regelbasierter Optimierer in Reinkultur, erzeugt aus einer gegebenen Anfrage in fünf Schritten einen Ausführungsplan (s. Abb. D.6 a) [63]. Die Anfragesprache ist gleich der verwendeten deskriptiven Algebra (*descriptive algebra*). Diese ist eine um geometrische Operatoren erweiterte relationale Algebra. Als zusätzliche Erweiterung enthält sie die Möglichkeit, Ausdrücke an Variablen zu binden. Ein Auswertungsplan wird durch einen Ausdruck der Ausführungsalgebra (*executable algebra*) dargestellt. Die Ausführungsalgebra beinhaltet im wesentlichen verschiedene Implementierungen der deskriptiven Algebra und Scan-Operationen. Die Trennung zwischen deskriptiver Algebra und Ausführungsalgebra ist strikt, das heißt, es kommen keine gemischten Ausdrücke vor (außer während der expliziten Konvertierung (Schritt 4)).

Die Schritte 1 und 3 sind durch feste Algorithmen implementiert. Während der

Normalisierung (Schritt 1) werden Variablenvorkommen durch die an sie gebundenen Ausdrücke ersetzt. Dies ist notwendig, um das Optimierungspotential vollständig erschließen zu können. Schritt 3 führt für konstante Ausdrücke Variablen ein. Die entspricht der Entschachtelung von Anfragen vom Typ N und A (s. Kapitel **??** und [483]). Die Behandlung von gemeinsamen Teilausdrücken ist noch nicht implementiert, aber für Schritt 3 vorgesehen.

Die Schritte 2, 4 und 5 sind regelbasiert. Zur Formulierung der Regeln wird eine Regelbeschreibungssprache (*rule description language*) verwendet. Die Beschreibungen der Regeln werden in einer Datei abgelegt. Innerhalb der Datei werden Regeln zu Gruppen (*sections*) zusammengefaßt. Diese Gruppen werden nacheinander angewandt. Daraus ergeben sich auch für einen Schritt mehrere kleinere Schritte. Beispielsweise ist der Schritt 2 im OPTEX-Optimierer für Gral in vier Teilschritte unterteilt:

1. Dekomposition von Selektionen mit komplexen Selektionsprädikaten in eine Folge von Selektionen mit einfachen Selektionsprädikaten und Zerlegung von Verbundoperationen in eine Folge von Selektionen und Kreuzprodukten.

2. Eigentlicher IMPROVING Schritt (siehe unten).

3. Teilausdrücke bestehend aus einer Selektion und einem unmittelbar folgenden Kreuzprodukt werden in Verbundoperationen umgewandelt.

4. Bestimmung einer Ordnung zwischen den Verbundoperationen und Kreuzprodukten. Dabei werden Kreuzprodukte zum Schluß ausgeführt und kleine Relationen zuerst verbunden.

Jeder Gruppe wird eine von drei in Gral implementierten Suchstrategien zugeordnet.

STANDARD Führt solange alle Regeln einer Gruppe aus, bis keine Regel mehr anwendbar ist. Es werden keine Vorkehrungen getroffen, um Endlosschleifen zu verhindern. Die Regeln müssen also dementsprechend formuliert werden. Diese Strategie kann für Schritte 2 und 5 verwendet werden.

IMPROVING Diese Strategie unterstützt algebraische Optimierung in der deskriptiven Algebra (Schritt 2). Das Ziel ist hierbei eine gute Ordnung der algebraischen Operatoren zu erlangen. Hierzu wird eine partielle Ordnung der algebraischen Operatoren gemäß ihrer Kosten definiert (s. Abb. D.6 b) für ein Beispiel). Die IMPROVING-Strategie versucht dann die hierdurch definierte Ordnung in einem gegebenen Ausdruck zu erreichen. Hierzu wird sie zunächst rekursiv auf alle Teilausdrücke eines Ausdrucks angewendet. Regeln zur Umformung werden dann angewendet, wenn dadurch eine höhere Kohärenz der Operatorfolge im Ausdruck mit der der Operatorkostenhierarchie erreicht werden kann. Dies entspricht einem Bubble-sort auf dem Ausdruck. Ausdrücke mit der kleinsten Anzahl von *runs* werden bevorzugt. Dabei ist ein *run* eine Folge von Operatoren innerhalb des zu optimierenden Ausdrucks, dessen Operatoren gemäß der Operatorkostenhierarchie geordnet sind.

TRANSLATION Regelgruppen mit dieser Strategie werden während der Übersetzung von der deskriptiven Algebra in die Ausführungsalgebra angewendet (Schritt

4). Jede Regel beschreibt dabei die Übersetzung eines einzelnen deskriptiven Operators in einen Ausdruck der Ausführungsalgebra, also einen, der keine deskriptiven Operatoren enthalten darf. Die Übersetzung erfolgt lokal. Für Parameter, also beispielsweise Selektions- und Verbundprädikate, können Regeln angegeben werden, die einen Suchraum für die Reorganisation des Parameters erlauben. Hiermit kann man beispielsweise alle Permutationen einer Konjunktion erzeugen. Die Suchstrategie für die Parameterbestimmung ist erschöpfend und trägt Vorsorge, daß keine Zyklen auftreten. Eine Auswahl kann mittels des *valuation*-Eintrags in den Regeln getroffen werden. Dieser kann beispielsweise Kosten repräsentieren. Dementsprechend werden dann Regeln mit der kleinsten *valuation* bevorzugt. Jede für einen Parameter generierte Darstellung wird übersetzt.

Die Syntax für eine Regel ist

specification
definition
RULE
pattern
$\rightarrow$ result$_1$ valuation$_1$ if condition$_1$
...
$\rightarrow$ result$_n$ valuation$_n$ if condition$_n$

wobei

specification von der Form

SPEC spec$_1$,...,spec$_n$

ist. Dabei sind die spec$_i$ Range-Spezifikationen wie beispielsweise $op_i\ in\ <OpSet>$.

definition Variablen definiert (bspw. für Attributsequenzen). In Gral existieren verschiedene Sorten von Variablen für Attribute, Operationen, Relationen etc.

pattern ein Muster in Form eines Ausdrucks ist, der Variablen und Konstanten enthalten kann. Der Ausdruck kann ein Ausdruck der deskriptiven Algebra oder der Ausführungsalgebra sein.

condition$_i$ eine Bedingung ist. Diese Bedingung ist ein allgemeiner boolescher Ausdruck. Spezielle Prädikate wie *ExistsIndex* (existiert ein Index für eine Relation?) werden von Gral zur Verfügung gestellt.

result$_i$ wiederum ein Ausdruck ist, der das Ergebnis der Regel beschreibt.

valuation$_i$ ist ein arithmetischer Ausdruck, der einen numerischen Wert zurückliefert. Dieser kann in einer (Gral unterstützt mehrere) Auswahlstrategie herangezogen werden: Es wird die Regel mit der kleinsten *valuation* bevorzugt.

Die Auswertung einer Regel erfolgt standardmäßig. Sei E der Ausdruck auf den die Regel angewendet werden soll.

if $\exists$ Substitution σ, Unterausdruck E' von E mit $E'\sigma = pattern$
and $\forall 1 \leq i \leq j$: $\neg$ condition$_i$
and condition$_j$
then ersetzte E' in E durch $result_j\sigma$

Der Gral-Optimierer ist ein reiner regelbasierter Optimierer, der den Transformationsansatz verfolgt. Dementsprechend treffen alle vorher identifizierten Nachteile derselben zu.

Zu bemängeln sind im einzelnen folgende Punkte:

- Es erfolgt keine frühzeitige Bewertung der Alternativen.
- Die Suchstrategien sind fest eingebaut und nicht sonderlich ausgefeilt.
- Der Einbau von hochspezialisierten Algorithmen, die besondere Optimierungstechniken repräsentieren, ist schwierig, wenn nicht unmöglich.
- Eine Bestimmung der Verbundreihenfolge gemäß eines komplexeren Algorithmus ist nicht möglich.
- Da die Übersetzung in die Ausführungsalgebra lokal ist und keine Annotationen zugelassen sind, können vorhandene Sortierreihenfolgen nur schwer ausgenutzt werden.

Es wird nur eine Alternative der algebraischen Optimierung zur physischen Optimierung übergeben. Das kann zu Fällen führen, in denen der Optimierer niemals das Optimum finden kann. Wenngleich dies auch im allgemeinen nicht immer möglich ist, so sollte jedoch diese Eigenschaft nicht inhärent sein.

Positiv zu vermerken ist, daß für IMPROVING und TRANSLATION der Aufwand für das Pattern-matching vermutlich gering gehalten werden kann.

D.1.10 Lambda-DB

`http://lambda.uta.edu/lambda-DB/manual/overview.html`

D.1.11 Lanzelotte in short

Query Language Der Lanzelotte-Optimierer verwendet keine spezielle Anfragesprache. Ausgangspunkt der Betrachtungen sind sog. Anfragegraphen (request graphs, query graphs). Einzelheiten stehen in meiner Ausarbeitung. In einem Papier ([512]) wird gezeigt wie man von einer Regelsprache (RDL) zu Anfragegraphen kommt.

Internal Representation Die interne Repraesentation einer Anfrage ist der Class Connection Graph. Dort enthalten sind die Datenbankobjekte (Extensionen), die in der Anfrage referenziert werden aus der Sicht des physikalischen Schemas und die in der Anfrage bedeutsamen Beziehungen zwischen diesen Extensionen (Joins, Attributpfade, Selektionen).

Query Execution Plans QEPs werden als (deep) processing trees repraesentiert.

Architecture Der Lanzelotte-Optimierer ist regelbasiert.

Transformation versus generation Lanzelotte bietet Regeln fuer beide Spielarten. Sie unterscheidet enumerative search (Generierung), randomized search (Transformation) und genetic search (Transformation).

Control/Search-Strategy Lanzelotte versucht von den Einzelheiten der verwendeten Stategien zu abstrahieren und stellt eine erweiterbare Optimierung vor, die die Einzelheiten ueberdeckt. Die tatsaechlich zu einem bestimmten Zeitpunkt verwendete Strategie wird durch "assertions" bestimmt. (Dazu steht nicht viel in den Papieren, vielleicht meint sie auch die Bedingungsteile der Regeln)

Cost Model Ziemlich aehnlich dem, das wir verwenden. Sie benutzt auch solche Sachen wie $card(C), size(C), ndist(A_i), fan(A_i), share(A_i)$. Einzelheiten stehen in meiner Ausarbeitung.

D.1.12 Opt++

wisconsin

D.1.13 Postgres

Postgres ist kein Objektbanksystem sondern fällt in die Klasse der erweiterten relationalen Systeme [815]. Die wesentlichen Erweiterungen sind

- berechenbare Attribute, die als Quel-Anfragen formuliert werden [813],
- Operationen [811],
- abstrakte Datentypen [810] und
- Regeln [814].

Diese beiden Punkte sollen uns jedoch an dieser Stelle nicht interessieren. Die dort entwickelten Optimierungstechniken, insbesondere die Materialisierung der berechenbaren Attribute, sind in der Literatur beschrieben [458, 394, 392, 393]. Unser Interesse richtet sich vielmehr auf eine neuere Publikation, in der eine Vorschlag für die Reihenfolgebestimmung von Selektionen und Verbundoperationen unterbreitet wird [406]. Diese soll im folgenden kurz vorgestellt werden. Zunächst jedoch einige Vorbemerkungen.

Wenn man eine Selektion verzögert, also nach einem Verbund ausführt, obwohl dies nicht notwendig wäre, so kann es passieren, daß das Selektionsprädikat auf mehr Tupeln ausgewertet werden muß. Es kann jedoch nicht passieren, daß es auf mehr verschiedenen Werten ausgeführt werden muß. Im Gegenteil, die Anzahl der Argumentewerte wird durch einen Verbund im allgemeinen verkleinert. Cached man also die bereits errechneten Werte des Selektionsprädikates, so wird die Anzahl der Auswertungen des Selektionsprädikates nach einem Verbund zumindest nicht größer. Die Auswertung wird dann durch ein Nachschlagen ersetzt. Da wir hier nur teure Selektionsprädikate betrachten, ist ein Nachschlagen sehr billig gegenüber der Auswertung. Die Kosten für das Nachschlagen können sogar vernachläßigt werden. Es bleibt

das Problem der größe des Caches. Liegt Eingabe sortiert nach den Argumenten des Selektionsprädikates vor, so kann der die größe des Caches unter Umständen auf 1 reduziert werden. Er erübrigt sich ganz, wenn man eine indirekte Repräsentation des Verbundergebnisses verwendet. Eine mögliche indirekte Repräsentation ist in Abbildung **??** dargestellt, wobei die linke der abgebildeten Relationen die Argumente für das betrachtete Selektionsprädikat enthalte.

Für jedes Selektionsprädikat $p(a_1, \ldots, a_n)$ mit Argumenten a_i bezeichne c_p die Kosten der Auswertung auf einem Tupel. Diese setzen sich aus CPU- und I/O-Kosten zusammen (s. [406]). Ein *Plan* ist ein Baum, dessen Blätter *scan*-Knoten enthalten und dessen innere Knoten mit Selektions- und Verbundprädikaten markiert sind. Ein *Strom* in einem Plan ist ein Pfad von einem Blatt zur Wurzel. Die zentrale Ide ist nun die Selektions- und Verbundprädikate nicht zu unterscheiden, sondern gleich zu behandeln. Dabei wird angenommen, daß alle diese Prädikate auf dem Kreuzprodukt aller Relationen der betrachteten Anfrage arbeiten. Dies erfordert eine Anpassung der Kosten. Seien $a_1, \ldots, a_n$ die Relationen der betrachteten Anfrage und p ein Prädikat über den Relationen $a_1, \ldots, a_k$. Dann sind die *globalen Kosten* von p wie folgt definiert:

$$C(p) = \frac{c_p}{\prod_{i=k+1}^{n} |a_i|}$$

Die globalen Kosten berechnen die Kosten der Auswertung des Prädikates über der gesamten Anfrage. Hierbei müssen natürlich diejenigen Relationen herausgenommen werden, die das Prädikat nicht beeinflussen. Zur Illustration nehme man an, p sei ein Selektionsprädikat auf nur einer Relation a_1. Wendet man p direkt auf a_1 an, so entstehen die Kosten $c_p * |a_1|$. Im vereinheitlichten Modell wird angenommen, daß jedes Prädikat auf dem Kreuzprodukt aller in der Anfrage beteiligten Relationen ausgewertet wird. Es entstehen also die Kosten $C(p) * |a_1| * |a_2| * \ldots * |a_n|$. Diese sind aber gleich $c_p * |a_1|$. Dies ist natürlich nur unter der Verwendung eines Caches für die Werte der Selektionsprädikate korrekt. Man beachte weiter, daß die Selektivität $s(p)$ eines Prädikates p unabhängig von der Lage innerhalb eines Stroms ist.

Der *globale Rang* eines Prädikates p ist definiert als

$$\mathit{rank}(p) = \frac{s(p)}{C(p)}$$

Man beachte, daß die Prädikate innerhalb eines Stroms nicht beliebig umordbar sind, da wir gewährleisten müssen, daß die von einem Prädikat benutzten Argumente auch vorhanden sein müssen. In [406] wir noch eine weitere Einschränkung vorgenommen: Die Verbundreihenfolge darf nicht angetastet werden. Es wird also vorausgesetzt, daß eine optimale Verbundreihenfolge bereits bestimmt wurde und nur noch die reinen Selektionsprädikate verschoben werden dürfen.

Betrachtet man zunächst einmal nur die Umordnung der Prädikate auf einem Strom, so erhält man bedingt durch die Umordbarkeitseinschränkungen das *Sequentialisierungsproblem mit Vorrangbedingungen* für das Algorithmus mit Laufzeit $O(nlogn)$ (n ist die Stromlänge) eine optimale Lösung bekannt ist [604].

Das in [406] vorgeschlagene Verfahren wendet diesen Algorithmus solange auf jeden Strom an, bis keine Verbesserung mehr erzielt werden kann. Das Ergebnis ist ein polynomialer Algorithmus, der die optimale Lösung garantiert. Dies jedoch nur unter der Einschränkung, daß die Kosten des Joins linear sind.

Damit sind wir bereits bei einem der Nachteile des Verfahrens: Die Kosten der Verbundoperation nicht mitunter nicht linear sondern sogar quadratisch. Ein weiterer Nachteil liegt in der Voraussetzung, daß die optimale Verbundreihenfolge schon bestimmt wurde, denn diese hängt wesentlich davon ab, an welcher Stelle die Selektionen eingebaut werden. Üblicherweise wird bei der Bestimmung der optimalen Verbundreihenfolge vorausgesetzt, daß alle Selektionsprädikate soweit wie möglich nach unten verschoben werden. Dies ist jedoch jetzt nicht mehr der Fall. Es ist also notwendig die Selektionsprädikatmigration in die Joinreihenfolgebestimmung zu integrieren. Nur dann kann man auf gute Ergebnisse hoffen. Die Integration mit einem Ansatz des dynamischen Programmierens ist problematisch, da dort Lösungen verworfen werden, die unter Umständen zur Optimalen Lösung führen, wenn ein Selektionsprädikat nicht ganz nach unten durchgeschoben wird [406].

Eine Teillösung wird dort auch angedeutet. Ist der Rang eines Selektionsprädikates größer als jeder Rang jedes Plans einer Menge von Verbunden, so ist das Selektionsprädikat in einem optimalen Baum oberhalb all dieser Verbundoperationen plaziert. Ein entsprechender Algorithmus hat aber, wenn er beispielsweise nur Left-deep-trees erzeugt, eine Worst-case-Komplexität von $O(n^4 n!)$.

D.1.14 Sciore & Sieg

Die Hauptidee von Sciore und Sieg ist es, die Regelmenge in Module zu organisieren und jedem Modul eine eigene Suchstrategie, Kostenberechnung und Regelmenge zuzuordnen. Module können andere Module explizit aufrufen, oder implizit ihre Ausgabemenge an das nächste Modul weiterleiten.

D.1.15 Secondo

Gueting

D.1.16 Squiral

Der erste Ansatz eines regelbasierten Optimierers, Squiral, kann auf das Jahr 1975 zurückgeführt werden [794]. Man beachte, daß dieses Papier vier Jahre älter ist als das vielleicht am häufigsten zitierte Papier über den System R Optimierer [750], der jedoch nicht regelbasiert, sondern fest verdrahtet ist.

Abbildung D.7 gibt einen Überblick über den Aufbau von Squiral. Nach der syntaktischen Analyse liegt ein Operatorgraph vor. Dieser ist in Squiral zunächst auf einen Operatorbaum beschränkt. Zur Behandlung von gemeinsamen Teilausdrücken wird das Anlegen von temporären Relationen, die den gemeinsamen Teilausdrücken entsprechen, vorgeschlagen. Diese temporären Relationen ersetzen dann die gemeinsamen Teilausdrücke. Dadurch ist es möglich, sich auf Operatorbäume zu beschränken.

Der Operatorbaum wird dann in einen optimierten Operatorbaum transformiert. Hierzu werden Regeln, die den algebraischen Gleichungen entsprechen, verwendet. Die Anwendung dieser Transformationsregeln ist rein heuristisch gesteuert. Die Heuristik selber ist in den Transformationsanwendungsregeln abgelegt. Eine dieser Regeln sagt beispielsweise, daß Projektionen nur dann nach unten geschoben werden, wenn

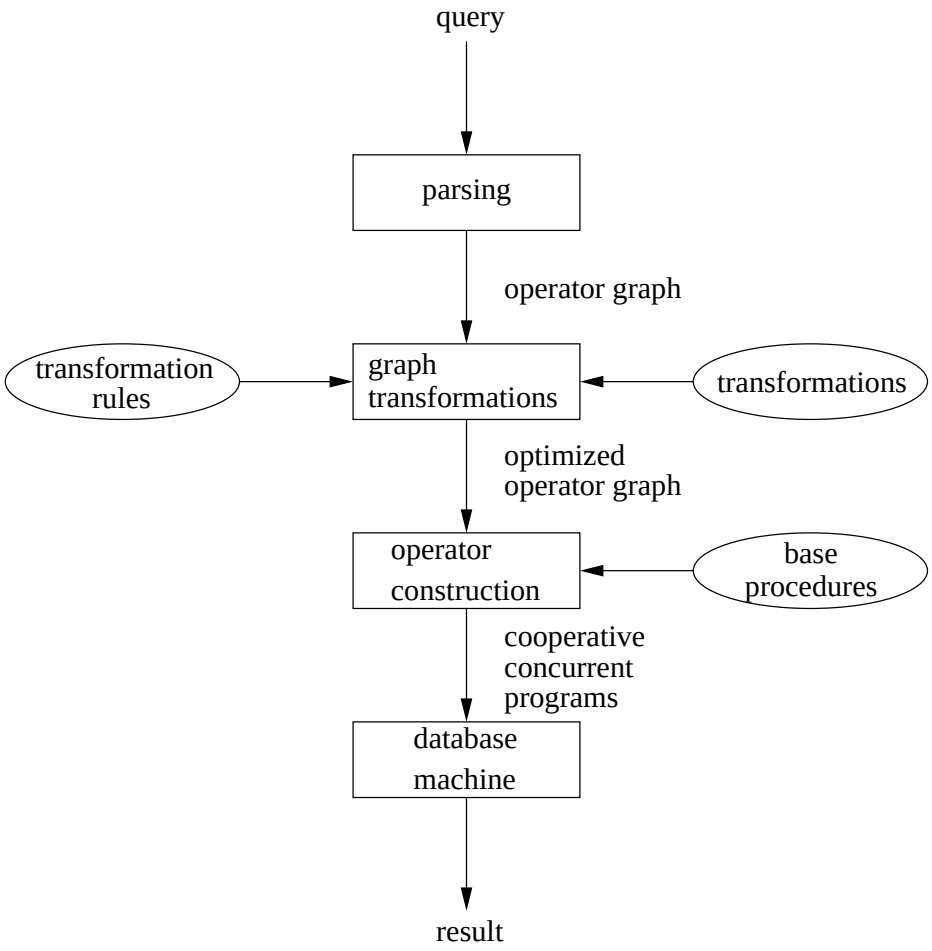


Figure D.7: Die Squiralarchitektur

die Operation, über die die Projektion als nächstes geschoben werden soll, keine Verbundoperation ist. Neben den Standardregeln, die das Vertauschen von relationalen Operatoren ermöglichen, gibt es Regeln, die es erlauben, relationale Ausdrücke in komplexe boolesche Ausdrücke, die dann als Selektionsprädikate Verwendung finden, zu überführen. Dies ist der erste Vorschlag, nicht nur primitive Selektionsprädikate in Form von Literalen, sondern auch komplexere Ausdrücke mit booleschen Verknüpfungen zu verwenden. Auf die Optimierung dieser Ausdrücke wird jedoch nicht weiter eingegangen.

Die wesentliche Aufgabe der Operatorkonstruktion ist die Auswahl der tatsächlichen Implementierungen der Operatoren im Operatorgraph unter optimaler Ausnutzung gegebener Sortierreihenfolgen. Auch diese Phase der Optimierung ist in Squiral nicht kostenbasiert. Sie wird durch zwei Durchläufe durch den Operatorgraphen realisiert. Der erste Durchlauf berechnet von unten nach oben die möglichen Sortierungen, die ohne zusätzlichen Aufwand möglich sind, da beispielsweise Relationen schon sortiert sind, und vorhandene Sortierungen durch Operatoren nicht zerstört werden. Im zweiten Durchlauf, von oben nach unten, werden Umsortierungen nur dann vorgenommen, wenn keine der im ersten Durchlauf berechneten Sortierungen eine effiziente Imple-

mentierung des zu konvertierenden Operators erlaubt. Beide Durchläufe sind mit Regelsätzen spezifiziert. Es ist bemerkenswert, daß die Anzahl der Regeln, 32 für den Aufwärtspaß und 34 für den Abwärtspaß, die Anzahl der Regeln für die Transformationsphase (insgesamt 7 Regeln), bei weitem übertrifft. Auch die Komplexität der Regeln ist erheblich höher.

Beide für uns interessante Phasen, die Operatorgraphtransformation und Operatorkonstruktion, sind mit Regeln spezifiziert. Es ist jedoch in beiden Phasen kein Suchprozeß nötig, da die Regeln alle Fälle sehr gezielt auflisten und somit einen eindeutigen Entscheidungsbaum beschreiben. Eine noch minutiösere Unterscheidung für die Erzeugung von Ausführungsplänen in der Operatorkonstruktionsphase gibt es nur noch bei Yao [923]. Diese haben auch den Vorteil, durch Kostenrechnungen belegt zu sein.

Da die Regeln in ihren Prämissen die Heuristik ihrer Anwendung mit kodieren und keine eigene Suchfunktion zur Anwendung der Regeln existiert, ist die Erweiterbarkeit sehr schwierig. Das Fehlen jeglicher Kostenbewertung macht eine Evaluation der Alternativen unmöglich. Daher ist es auch schwer, die einzelnen Komponenten des Optimierers, nämlich die Regeln, zu bewerten, zumal der transformierende Ansatz gewählt wurde. Der Forderung nach Vorhersagbarkeit und stetiger Leistungsabfall wird in diesem Ansatz ebenfalls nicht nachgegangen.

D.1.17 System R and System R*

D.1.18 Starburst and DB2

Starburst [257, 378] liegt ein erweiterbares relationales Datenmodell zugrunde. Die Anfragebearbeitung ist wie in System R und System R* in die zwei Schritte Anfrageübersetzung und -ausführung zergliedert [379]. Wir interessieren uns für den ersten Schritt, die Anfrageübersetzung. Einen Überblick gibt Abbildung D.8. Nach der standardmäßigen Zerteilung liegt die Anfrage in der internen Darstellung QGM (Query Graph Model) vor. QGM ist an die Anfragesprache Hydrogen (ähnlich SQL) von Starburst angelehnt. Der wichtigste Grundbaustein von QGM ist der *select*-Operator. Dieser enthält eine Projektionsliste und das Anfrageprädikat in Graphform. Die Knoten sind markiert und referenzieren (gespeicherte) Relationen oder weitere QGM-Operatoren. Die Markierung ist entweder ein Quantor ($\forall$, $\exists$) oder die Mengenerzeugermarkierung (F). Knoten, die mit F markiert sind, tragen zur Erzeugung des Ergebnisses eines Operators bei, die Quantorenmarkierungen zu dessen Einschränkung. Die Kanten sind mit den Prädikaten markiert. Es ergeben sich also Schleifen für nur eine Relation betreffende Prädikate. Weitere Operatoren sind *insert, update, intersection, union* und *group-by*. Daneben wird die QGM-Repräsentation einer Anfrage mit Schemainformation und statistischen Daten angereichert. Sie dient also auch als Sammelbecken für alle die Anfrage betreffende Information.

Die QGM-Repräsentation dient der Anfragetransformation (Abb. D.8) als Ausgangspunkt. Die Anfragetransformation generiert zu einer QGM-Repräsentation verschiedene äquivalente QGM-Repräsentationen. Die Anfragetransformation läßt sich, abgesehen von den Darstellungsunterschieden von QGM und Hydrogen, als eine Variante der Source-level-Transformationen ansehen. Sie wird regelbasiert implementiert, wobei C die Regelsprache ist. Eine Regel besteht aus 2 Teilen, einer Bedingung und

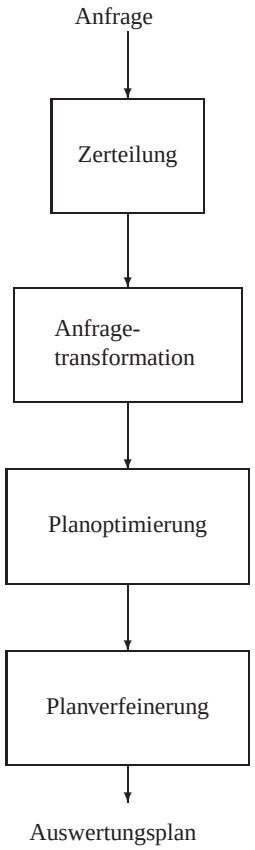


Figure D.8: Starburst Optimierer

einer Aktion. Jeder Teil wird durch eine C-Prozedur beschrieben. Dadurch erübrigt sich die Implementierung eines allgemeinen Regelinterpreters mit Pattern-matching. Regeln können in Gruppen zusammengefaßt werden. Der aktuelle Optimierer umfaßt drei Klassen von Regeln:

1. Migration von Prädikaten
2. Migration von Projektionen
3. Verschmelzung von Operationen

Für die Ausführung der Regeln stehen drei verschiedene Suchstrategien zur Verfügung:

1. sequentiell,
2. prioritätsgesteuert und
3. zufällig, gemäß einer gegebenen Verteilung.

Die Teilgraphen der QGM-Repräsentation, auf die Regeln anwendbar sind, können entweder durch eine depth-first oder eine breadth-first Suche bestimmt werden. Falls mehrere alternative QGM-Repräsentationen existieren (was meistens der Fall ist), wird

ein *Choose*-Operator [351] verwendet, der die verschiedenen QGMs in einen QGM zusammenbaut. Die nachfolgende Phase wählt dann kostenbasiert einen dieser alternativen QGMs aus. Dies ist nicht zwingend, die Auswahl kann auch erst zur Auswertungszeit stattfinden. Begründet wird dieses Vorgehen damit, daß keine Kosten für QGMs berechnet werden können, und somit keine Bewertung eines QGMs stattfinden kann. Wie die Autoren selbst anmerken, ist dieser Umstand sehr mißlich, da keine Alternativen verworfen werden können. Sie kündigen daher Untersuchungen an, die Transformation (Schritt 2) mit der Planoptimierung (Schritt 3) zu verschmelzen. Um eine gewisse Kontrolle über das Verhalten der Transformation zu haben, kann diesem Schritt ein "budget" mitgegeben werden, nach dessen Ablauf der Schritt beendet wird. Die genaue Funktionsweise des "budget" ist leider nicht erläutert.

Der Schritt der Planoptimierung (s. Abb. D.8) kann mit der bisherigen Optimierung verglichen werden. Sie arbeitet regelbasiert, benutzt aber nicht den transformierenden, sondern den generierenden Ansatz [553]. Aus Basisoperationen – LOLEPOPs (LOw-LEvel Plan OPerator) genannt – werden mit (grammatischen) Regeln – STARs (strategy alternative rules) genannt – (alternative) Auswertungspläne erzeugt. LOLEPOPs entstammen der um SCAN, SORT und ähnliche physische Operatoren angereicherten relationalen Algebra. Ein Auswertungsplan ist dann ein Ausdruck von geschachtelten Funktionsaufrufen, wobei die Funktionen den LOLEPOPs entsprechen.

Ein STAR definiert ein benanntes parametrisiertes Objekt, das einem Nichtterminalsymbol entspricht. Er besteht aus einer Menge von alternativen Definitionen, die jede aus einer Bedingung für die Anwendbarkeit und der Definition eines Plans bestehen. Der generierte Plan kann LOLEPOPs (entsprechen Terminalsymbolen) und STARs referenzieren. Ein rootSTAR entspricht dem Startsymbol der Grammatik. STARs ähneln den Regeln, die in Genesis nicht nur für den Optimierer, sondern für das ganze DBMS eingesetzt werden, um alternative Implementierungen zu erhalten [55, 53, 57, 56, 58]. Um erzeugte Alternativen für einen Plan zusammenzusetzen und zu verhindern, daß diese Alternativen die Anzahl der Pläne in denen diese vorkommen, vervielfachen, wird ein Glue-Mechanismus eingesetzt. Dieser hat den *Choose*-Operator als Wurzel. Darunter hängen dann Alternativen, die beispielsweise einen Strom mit gewissen Eigenschaften (Sortierung, Lokation) erzeugen. Von diesen Alternativen werden nur diejenigen betrachten, die die geringsten Kosten bei gleichen Eigenschaften haben [525]. Die Kosten beziehen sich dabei immer nur auf den bisher erreichten Teilplan.

Der Aufbau eines Auswertungsplanes erfolgt Bottom-up. Die Menge der anwendbaren STARs wird in einer ToDo-Liste gehalten. Diese ist eine sortierte Liste. Hiermit können dann verschiedene Suchstrategien implementiert werden, indem verschiedene Sortierungen für die ToDo-Liste Verwendung finden [525]. Ein Vorteil des STAR-Ansatzes ist die Vermeidung von Pattern-matching. Dies erlaubt es, die STARs zu interpretieren [525].

Die Beurteilung der Erweiterbarkeit ist sehr schwierig. Zum einen handelt es sich um einen erweiterbaren Optimierer, da sowohl LOLEPOPs als auch STARs hinzugefügt werden können. Der Glue-Mechanismus kann ebenfalls spezifiziert werden, ohne in die Implementierung einzugreifen. Das Problem ist lediglich die Komplexität dieser Änderungen. Man kann diesen Ansatz daher vielleicht als bedingt erweiterbar kennzeichnen.

Eine Trennung der Optimierung in verschiedene Phasen wirft die damit verbunde-

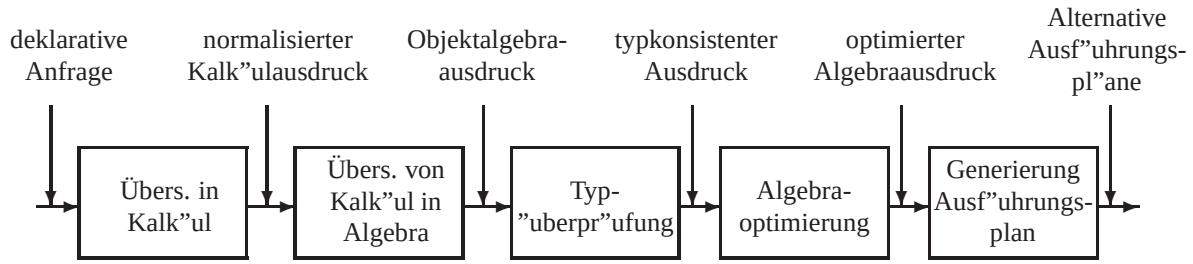


Figure D.9: Der Optimierer von Straube

nen Probleme auf. Wie oben bereits angeführt, kündigen die Autoren weitere Untersuchungen an, um eine Verschmelzung der Phasen zu ermöglichen. Da keine Alternativen verworfen werden, ist es potentiell möglich, den optimalen Auswertungsplan zu errechnen. Schwer zu sehen ist jedoch, wie ein stetiger Leistungsabfall zu realisieren ist. Gleiches gilt für die Evaluierbarkeit der einzelnen Komponenten (STARs).

More on Starburst can be found in [661, 662].

D.1.19 Der Optimierer von Straube

In seiner Dissertation stellt Straube den von ihm entwickelten Optimierer dar [821]. Die Ergebnisse dieser Arbeit flossen in eine Reihe von Veröffentlichungen ein [851, 818, 819, 817, 820]. Der Aufbau des Optimierers ist in Abbildung D.9 skizziert. Eine Anfrage wird zunächst in den Objektkalkül übersetzt und von dort in die Objektalgebra. Hier findet dann zunächst eine Typüberprüfung statt. Danach beginnt die eigentliche Optimierung. Diese besteht aus zwei Phasen, der algebraischen Optimierung und der Generierung des Ausführungsplans.

Die erste Phase, die algebraische Optimierung, folgt dem Tranformationsparadigma. Die algebraischen Ausdrücke werden mit Hilfe von Regeln in äquivalente algebraische Ausdrücke transformiert. Straube beschränkt sich dabei im wesentlichen auf die Formulierung der Regeln. Für die Abarbeitung der Regeln schlägt er lediglich die Verwendung des Exodus-Optimierergenerators [347] oder des Anfrageumformers von Starburst [402] vor.

Die zweite Phase, die Generierung der Ausführungspläne, ist nicht regelbasiert. Ihr liegt eine sogenannte Ausführungsplanschablone zu Grunde. Sie ist vergleichbar mit einem Und/Oder-Baum, der alle möglichen Ausführungspläne implizit beinhaltet. Zur Generierung eines konkreten Ausführungsplans wird der durch die Ausführungsschablone aufgespannte Suchraum vollständig durchsucht. Der billigste Ausführungsplan kommt dann zur Abarbeitung.

Da ein regelbasierten Ansatz für die erste Phase gewählt wurde und die Verwendung des Exodus-Optimierergenerators oder des Starburst-Anfrageumformers vorgeschlagen wird, verweisen wir für die Bewertung dieser Phase auf die entsprechenden Abschnitte.

Die zweite Phase ist voll auskodiert und damit schlecht erweiterbar. Eine frühe Bewertung der Alternativen ist nicht ausgeschlossen, wird aber nicht vorgenommen. Ein vollständiges Durchsuchen verhindert natürlich auch den stetigen Leistungsabfall des Optimierers.

Erschwerend für den gewählten Ansatz kommt die Zweiphasigkeit hinzu. Es ist schwierig zu sehen, wie eine phasenübergreifende Kontrolle auszusehen hat, die zumindest potentiell Optimalität gewährleistet. Die Evaluierbarkeit einzelner Komponenten des Optimierers ist nicht möglich.

D.1.20 Other Query Optimizer

Neben den in den vorangehenden Abschnitten erwähnten Optimierern gibt es noch eine ganze Reihe anderer, die aber nicht im einzelnen vorgestellt werden sollen. Erwähnt werden sollen noch die Systeme Probe [218, 217, 629] und Prima [398, 396]. Der Schwerpunkt bei der Optimierung liegt im Primasystem auf dem dynamischen Zusammenbau von Molekülen. Es wäre zu untersuchen, ob ein Assembly-Operator (s. [471]) hier von Nutzen wäre. Besonders erwähnenswert ist noch eine Arbeit, die Optimierungsmöglichkeiten für die Datenbankprogrammiersprache FAD vorstellt [856]. Diese Arbeit stellt einen ersten Schritt in Richtung eines Optimierers für eine Generalpurpose-Programmiersprache dar. Ein wesentlicher Punkt ist dabei, daß auch ändernde Operationen optimiert werden. Der Optimierer ist in zwei Module (RWR and OPT) eingeteilt. RWR ist ein Sprachmodul, das die Übersetzung von FAD in ein internes FAD vornimmt. Immer wenn RWR einen Ausdruck erkennt, der in der vom Optimierer OPT bearbeitbaren Sprache ausgedrückt werden kann, so wird dieser an den Optimierer weitergegeben und dort optimiert. Es wird Exhaustive search als Suchstrategie für den Optimierer vorgeschlagen.

Im erweiterten O_2-Kontext wurde das Zerteilen von Pfadausdrücken weiter untersucht [184]. Es werden die Vorteile einer typisierten Algebra für diese Zwecke herausgearbeitet. Eine graphische Notation sorgt für eine anschauliche Darstellung. Ihr besonderes Augenmerk richten die Autoren auf die Faktorisierung gemeinsamer Teilausdrücke. Einige der Ersetzungsregeln sind aus [772] und [771] entnommen und werden gewinnbringend eingesetzt, so beispielsweise die Ersetzung von Selektionen durch Verbundoperatoren.

Ebenfalls erwähnt wurden bereits die Arbeiten im Orion-Kontext [48, 49, 482], die sich auf die Behandlung von Pfadausdrücken konzentrieren. Auch hier wurde ein funktionsfähiger Optimierer entwickelt.

Wie bereits erwähnt, stammt der erste regelbasierte Optimierer von Smith und Chang [794]. Doch erst die neueren Arbeiten führten zu einer Blüte des regelbasierten Ansatzes. Hier ist insbesondere die Arbeit von Freytag zu erwähnen, die diese Blüte mit initiierte [280]. Dort wird gezeigt, wie man mit Hilfe eines regelbasierten Ansatzes den Optimierer von System R [750] emulieren kann. Die Eingabe besteht aus einem Lisp-ähnlichen Ausdruck:

```
(select <proj-list>
        <sel-pred-list>
        <join-pred-list>
        <table-list>)
```

Die Projektionsliste besteht aus Attributspezifikationen der Form

```
<rel-name>.<attr-name>
```

Diese werden auch für die Selektionsprädikate und Joinprädikate verwendet. Die Algebra beinhaltet sowohl Operatoren der logischen als auch der physischen Algebra. Im einzelnen gibt es Scan-, Sort-, Projektions, Verbundoperatoren in einer logischen und verschiedenen physischen Ausprägungen. Die Erzeugung der Auswertungspläne wird in verschiedene Schritte unterteilt, die wiederum in Teilschritte zerlegt sind (siehe Abb. D.3).

Zunächst erfolgt die Übersetztung in die logische Algebra. Hier werden Scan-Operatoren um die Relationen gebaut und Selektionen, die nur eine Relation betreffen, in die Scan-Operatoren eingebaut. Der zweite Schritt generiert Zugriffspläne, indem der Scan-Operator durch einen einfachen File-Scan (FSCAN) ersetzt wird oder falls möglich, durch einen Index-Scan (ISCAN). Der dritte Schritt generiert zunächst verschiedene Verbundreihenfolgen und bestimmt anschließend die Verbundmethoden. Wie in System R wird zwischen dem Sortiere-und-Mische-Verbund und dem Verbund durch geschachtelte Schleifen unterschieden. Es werden keinerlei Aussagen über die Auswahl einer Suchstrategie gemacht. Ziel ist es vielmehr, durch die Modellierung des System R Optimierers mit Hilfe eines Regelsystems die prinzipielle Brauchbarkeit des regelbasierten Ansatzes nachzuweisen.

Man beachte auch die erwähnte Arbeit von Sciore und Sieg zur Modularisierung von regelbasierten Optimierern [746]. Die Hauptidee von Sciore und Sieg ist es, die Regelmenge in Module zu organisieren und jedem Modul eine eigene Suchstrategie, Kostenberechnung und Regelmenge zuzuordnen. Module können andere Module explizit aufrufen oder implizit ihre Ausgabemenge an das nächste Modul weiterleiten. Der erste Optimierer des GOM-Systems ist ebenfalls regelbasiert [474, 473]. Die gesamte Regelmenge wurde hier in Teilmengen ähnlich zu den Modulen organisiert. Die Steuerung zwischen den Teilmengen erfolgt durch ein heuristisches Netz, das angibt in welchen Fällen zu welcher weiteren Teilmenge von Regeln zu verzweigen ist. Die Strukturierung des Optimiererwissens steht auch in [566] im Vordergrund.

In diesem Zusammenhang, der Strukturierung von Optimierern und der Wiederverwendbarkeit einzelner Teile, sei noch einmal ausdrücklich auf die Arbeiten von Batory [56] aus dem Genesiskontext hingewiesen (s. auch Abschnitt D.1.7). Der dort leider ein wenig zu kurz kommende Aspekt der Wiederverwendbarkeit von Suchfunktionen wird in einer Arbeit von Lanzelotte und Valduriez [513] ausführlicher behandelt. Hier wurde eine Typhierarchie existierender Suchfunktionen entworfen und deren Schnittstellen vereinheitlicht. Die Suchfunktionen selbst wurden modularisiert. Weitere Arbeiten aus derselben Gruppe beschäftigen sich mit der Optimierung von objektorientierten Anfragen [512, 516], wobei hier die Behandlung von Pfaden im Vordergrund steht. Eine neuere Arbeit beschäftigt sich mit der Optimierung von rekursiven Anfragen im objektorientierten Kontext [514].

Viele kommerzielle Systeme besitzen eine Anfragesprache und einen Optimierer. Einer der wenigen Optimierer, die auch in der Literatur beschrieben werden, ist der von ObjectStore [628]. Durch die einfache Anfragesprache, die nur Teilmengenbestimmung erlaubt, und die strikte Verwendung von C-Semantik für boolesche Ausdrücke sind die meisten Optimierungsmöglichkeiten jedoch ausgeschlossen, und der "Optimierer" ist daher sehr einfach.

D.2 Commercial Query Compiler

D.2.1 The DB 2 Query Compiler

D.2.2 The Oracle Query Compiler

Oracle still provides two modes for its optimizer. Dependent on the user specified optimizer mode, a query is optimized either by the rule-based optimizer (RBO) or by the cost-based optimizer (CBO). The RBO is a heuristic optimizer that resembles the simple optimizer of chapter 2. Here we concentrate on the more powerful CBO. The user can also determine whether the optimizer should optimize for throughput or response time.

- nested loop join, nested loop outer join, index nested loop joins, sort merge join, sort merge outer join, hash joins, hash outer join, cartesian join, full outer join, cluster join, anti-joins, semi-joins, uses bitmap indexes for star queries
- sort group-by,
- bitmap indexes, bitmap join indexes
- index skip scans
- partitioned tables and indexes
- index-organized tables
- reverse key indexes
- function-based indexes
- SAMPLE clause in SELECT statement
- parallel query and parallel DML
- star transformations and star joins
- query rewrite with materialized views
- cost: considers CPU, I/O, memory
- access path: table scan, fast full index scan, index scan, ROWID scans (access ROW by ROWID), cluster scans, hash scans. [former two with prefetching] index scans:
  - index unique scan (UNIQUE or PRIMARY KEY constraints)
  - index range scan (one or more leading columns or key)
  - index range scan descending
  - index skip scan ($>$ 1 leading key values not given)
  - index full scan, index fast full scan
  - index joins (joins indexes with hash join, resembles index anding)

- bitmap joins (index anding/oring)
- cluster scan: for indexed cluster to retrieve rows with the same cluster id
- hash scan: to locate rows in a hash cluster

CBO: parsed quer –¿ [query transformer] –¿ [estimator] –¿ [plan generator] 1-16.
after parser: nested query blocks
simple rewrites:

- eliminate between
- elminate x in (c1 . . . cn) (also uses IN-LIST iterator as outer table constructor in a d-join or nested-loop join like operation.

query transformer:

- view merging
- predicate pushing
- subquery unnesting
- query rewrite using materialized views (cost based)

remaining subplans for nested query blocks are ordered in an efficient manner
plan generator:

- choose access path, join order (upper limit on number of permutations considered), join method.
- generate subplan for every block in a bottom-up fashion
- (> 1 for still nested queries and unmerged views)
- stop generating more plans if there already exists a cheap plan
- starting plan: order by their effective cardinality
- considers normally only left-deep (zig-zag) trees.
- single row joins are placed first (based on unique and key constraints.
- join statement with outer join: table with outer join operator must come after the other table in the condition in the join order. optimizer does not consider join orders that violate this rule.
- NOT IN (SELECT . . .) becomes a anti-join that is executed as a nested-loop join by default unless hints are given and various conditions are met which allow the transformation of the NOT IN uncorrelated subquery into a sort-merge or hash anti-join.
- EXISTS (SELECT . . .) becomes a semi-join. execution as index nested loops, if there is an index. otherwise a nested-loop join is used by default for EXISTS and IN subqueries that cannot be merged with the containing query unless a hint specifies otherwise and conditions are met to allos the transformation of the subquery into a sort-merge or hash semi-join.

- star query detection

cost:

- takes unique/key constraints into consideration
- low/high values and uniform distribution
- host variables: guess small selectivity value to favor index access
- histograms
- common subexpression optimization
- complex view merging
- push-join predicate
- bitmap access paths for tables with only B-tree indexes
- subquery unnesting
- index joins

rest:

- Oracle allows user hints in SQL statements to influence the Optimizer. for example join methods can be given explicitly

parameters:

- HASH_AREA_SIZE
- SORT_AREA_SIZE
- DB_FILE_MULTIBLOCK_READ_COUNT (number of prefetched pages)

statistics:

- table statistics
  number of rows, number of blocks, average row length
- column statistics
  number of distinct values, number of nulls, data distribution
- index statistics
  number of keys, (from column statistics?) number of leaf blocks, levels, clustering factor (collocation amount of the index block/data blocks, 3-17)
- system statistics
  I/O performance and utilization, cpu performance and utilization

generating statistics:

- estimation based on random data sampling
  (row sampling, block sampling)

- exact computation
- user-defined statistics collection methods

histograms:

- height-based histograms (approx. equal number of values per bucket)
- value-based histograms
  used for number of distinct values $\leq$ number of buckets
- support of index-only queries
- index-organized tables
- bitmap indexes (auch fuer null-werte `x <> const`)
- convert b-tree result RID lists to bitmaps for further bitmap anding
- bitmaps and count
- bitmap join index
- cluster tables (cluster rows of different tables on the same block)
- hash clusters
- hint: USE_CONCAT: OR ::= UNION ALL
- hint: STAR_TRANSFORMATION: see Oracle9i Database Concepts
- NOT IN ::= anti-join
- EXISTS ::= special join preserving duplicates and adding no phantom duplicates (semi-join) (5-27)
- continue 5-35

D.2.3 The SQL Server Query Compiler

Appendix E

Some Equalities for Binomial Coefficients

The following identities can be found in the book by Graham, Knuth, and Patashnik [352].

We use the following definition of binomial coefficients:

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!} & \text{if } 0 \le k \le n \\ 0 & \text{else} \end{cases} \tag{E.1}$$

We start with some simple identities.

$$\binom{n}{k} = \binom{n}{n-k} \tag{E.2}$$

$$\binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \tag{E.3}$$

$$k\binom{n}{k} = n\binom{n-1}{k-1} \tag{E.4}$$

$$(n-k)\binom{n}{k} = n\binom{n-1}{k} \tag{E.5}$$

$$(n-k)\binom{n}{k} = n\binom{n-1}{n-k-1} \tag{E.6}$$

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \tag{E.7}$$

$$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k} \tag{E.8}$$

The following identities are good for sums of binomial coefficients.

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \tag{E.9}$$

$$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1} \tag{E.10}$$

$$\sum_{k=0}^{n} \binom{m+k}{k} = \binom{m+n+1}{m+1} = \binom{m+n+1}{n} \tag{E.11}$$

$$\sum_{k=0}^{n} \binom{m-n+k}{k} = \binom{m+1}{n} \tag{E.12}$$

From Identities E.2 and E.11 it follows that

$$\sum_{k=0}^{m} \binom{k+r}{r} = \binom{m+r+1}{r+1} \tag{E.13}$$

For sums of products, we have

$$\sum_{k=0}^{n} \binom{r}{m+k}\binom{s}{n-k} = \binom{r+s}{m+n} \tag{E.14}$$

$$\sum_{k=0}^{n} \binom{l-k}{m}\binom{q+k}{n} = \binom{l+q+1}{m+n+1} \tag{E.15}$$

$$\sum_{k=0}^{n} \binom{l}{m+k}\binom{s}{n+k} = \binom{l+s}{l-m+n} \tag{E.16}$$

Bibliography

[1] K. Aberer and G. Fischer. Semantic query optimization for methods in object-oriented database systems. In *Proc. IEEE Conference on Data Engineering*, pages 70–79, 1995.

[2] S. Abiteboul. On Views and XML. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 1–9, 1999.

[3] S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, pages 117–138. Lecture PAGESs in Computer Science 361, Springer, 1987.

[4] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing restructuring. *Journal of Computer Science and Systems*, 33(3):361, 1986.

[5] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1):5–19, April 1997.

[6] S. Abiteboul and O. Duschka. Complexity of answering queries using materialized views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 254–263, 1998.

[7] A. Aboulnaga, A. Alameldeen, and J. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 591–600, 2001.

[8] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 181–192, 1999.

[9] A. Aboulnaga and J. Naughton. Accurate estimation of the cost of spatial selections. In *Proc. IEEE Conference on Data Engineering*, pages 123–134, 2000.

[10] A. Aboulnaga and J. Naughton. Building XML statistics for the hidden web. In *Int. Conference on Information and Knowledge Management (CIKM)*, pages 358–365, 2003.

[11] W. Abu-Sufah, D. J. Kuch, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. on Computers*, C-50(5):341–355, 1981.

[12] B. Adelberg, H. Garcia-Molina, and J. Widom. The STRIP rule system for efficiently maintaining derived data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 147–158, 1997.

[13] F. Afrati, M. Gergatsoulis, and T. Kavalieros. Answering queries using materialized views with disjunctions. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 435–452, 1999.

[14] F. Afrati, C. Li, and J. Ullman. Generating efficient plans for queries using views. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 319–330, 2001.

[15] F. Afrati and C. Papadimitriou. The parallel complexity of simple chain queries. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 210–?, 1987.

[16] D. Agrawal, A. Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 417–427, 1997.

[17] S. Agrawal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 506–521, 1996.

[18] R. Ahad, K. Bapa Rao, and D. McLeod. On estimating the cardinality of a database relation. *ACM Trans. on Database Systems*, 14(1):28–40, Mar. 1989.

[19] A. Aho, Y. Sagiv, and J. Ullman. Efficient optimization of a class of relational expressions. *ACM Trans. on Database Systems*, 4(4):435–454, 1979.

[20] A. Aho, Y. Sagiv, and J. Ullman. Equivalence of relational expressions. *SIAM J. on Computing*, 8(2):218–246, 1979.

[21] A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Trans. on Database Systems*, 4(3):297–314, 1979.

[22] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expression. *ACM Trans. on Database Systems*, 4(4):435–454, 1979.

[23] A.V. Aho, Y. Sagiv, and J.D. Ullman. Equivalence among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[24] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. IEEE Conference on Data Engineering*, pages 141–152, 2002.

[25] J. Albert. Algebraic properties of bag data types. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 211–219, 1991.

[26] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and optimization of compilers*, pages 1–30. Prentice Hall, 1971.

[27] J. Alsabbagh and V. Rahavan. Analysis of common subexpression exploitation models in multiple-query processing. In *Proc. IEEE Conference on Data Engineering*, pages 488–497, 1994.

[28] K. Alsabti, S. Ranka, and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 346–355, Athens, Greece, 1997.

[29] P. Alsberg. Space and time savings through large database compression and dynamic restructuring. In *Proc IEEE 63,8*, Aug. 1975.

[30] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava. Efficient algorithms for minimizing tree pattern queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 497–508, 2001.

[31] L. Amsaleg, M. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delay. In *4th Int. Conference on Parallel and Distributed Information Systems (PDIS)*, Palm Beach, Fl, 1996.

[32] O. Anfindsen. A study of access path selection in DB2. Technical report, Norwegian Telecommunication Administration and University of Oslo, Norway, Oct. 1989.

[33] G. Antoshenkov. Dynamic query optimization in RDB/VMS. In *Proc. IEEE Conference on Data Engineering*, pages 538–547, Vienna, Apr. 1993.

[34] G. Antoshenkov. Query processing in DEC Rdb: Major issues and future challenges. *IEEE Data Engineering Bulletin*, 16:42–52, Dec. 1993.

[35] G. Antoshenkov. Dynamic optimization of index scans restricted by booleans. In *Proc. IEEE Conference on Data Engineering*, pages 430–440, 1996.

[36] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Trans. on Software Eng.*, 9(1):57–68, 1983.

[37] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Trans. on Software Eng.*, 9(1):57–68, 1983.

[38] R. Ashenhurst. Acm forum. *Communications of the ACM*, 20(8):609–612, 1977.

[39] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975.

[40] M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. Mc Jones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[41] R. Avnur and J. Hellerstein. Eddies: Continiously adaptive query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2000.

[42] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 119–130, 2005.

[43] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 107–118, 2005.

[44] L. Baekgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM Trans. on Database Systems*, 20(2):111–148, 1995.

[45] T. Baer. Iperfex: A hardware performance monitor for Linux/IA32 systems. perform internet search for this or similar tools.

[46] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 60–71, 2004.

[47] F. Bancilhon and K. Koshafian. A calculus for complex objects. In *ACM Symp. on Principles of Database Systems*, pages 53–59, 1986.

[48] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. MCC Technical Report DB-188-87, MCC, Austin, TX 78759, June 1987.

[49] J. Banerjee, W. Kim, and K.-C. Kim. Queries in object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages 31–38, 1988.

[50] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 156–165, 1997.

[51] M. Bassiouni. Data compression in scientific and statistical databases. *IEEE Trans. on Software Eng.*, 11(10):1047–1058, 1985.

[52] D. Batory. On searching transposed files. *ACM Trans. on Database Systems*, 4(4):531–544, 1979.

[53] D. Batory. Extensible cost models and query optimization in Genesis. *IEEE Database Engineering*, 9(4), Nov 1986.

[54] D. S. Batory. Modeling the storage architecture of commercial database systems. *ACM Trans. on Database Systems*, 10(4):463–528, Dec. 1985.

[55] D. S. Batory. A molecular database systems technology. Tech. Report TR-87-23, University of Austin, 1987.

[56] D. S. Batory. Building blocks of database management systems. Technical Report TR-87-23, University of Texas, Austin, TX, Feb. 1988.

[57] D. S. Batory. Concepts for a database system compiler. In *Proc. of the 17nth ACM SIGMOD*, pages 184–192, 1988.

[58] D. S. Batory. On the reusability of query optimization algorithms. *Information Sciences*, 49:177–202, 1989.

[59] D. S. Batory and C. Gotlieb. A unifying model of physical databases. *ACM Trans. on Database Systems*, 7(4):509–539, Dec. 1982.

[60] D. S. Batory, T. Y. Leung, and T. E. Wise. Implementation concepts for an extensible data model and data language. *ACM Trans. on Database Systems*, 13(3):231–262, Sep 1988.

[61] L. Baxter. *TheComplexity of Unification*. PhD thesis, University of Waterloo, 1976.

[62] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Trans. on Database Systems (to appear)*, 1991.

[63] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Trans. on Database Systems*, 17(2):247–303, June 1992.

[64] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 72–88, 1990.

[65] C. Beeri and Y. Kornatzky. Algebraic optimization of object-oriented query languages. *Theoretical Computer Science*, 116(1):59–94, 1993.

[66] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.

[67] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):78–101, 1966.

[68] D. Bell, D. Ling, and S. McLean. Pragmatic estimation of join sizes and attribute correlations. In *Proc. IEEE Conference on Data Engineering*, pages 76–84, 1989.

[69] D. Beneventano, S. Bergamaschi, and C. Sartori. Description logic for semantic query optimization in object-oriented database systems. *ACM Trans. on Database Systems*, 28(1):1–50, 2003.

[70] K. Bennett, M. Ferris, and Y. Ioannidis. A genetic algorithm for database query optimization. Technical Report Tech. Report 1004, University of Wisconsin, 1990.

[71] K. Bennett, M. Ferris, and Y. Ioannidis. A genetic algorithm for database query optimization. In *Proc. 4th Int. Conf. on Genetic Algorithms*, pages 400–407, 1991.

[72] A. Bernasconi and B. Codenetti. Measures of boolean function complexity based on harmonic analysis. In M. Bonuccelli, P. Crescenzi, and R. Petreschi, editors, *Algorithms and Complexity (2nd. Italien Converence)*, pages 63–72, Rome, Feb. 1994. Springer (Lecture Notes in Computer Science 778).

[73] P. Bernstein, E. Wong, C. Reeve, and J. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Trans. on Database Systems*, 6(4):603–625, 1981.

[74] P. A. Bernstein and D. M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.

[75] P. A. Bernstein and N. Goodman. The power of inequality semijoin. *Information Systems*, 6(4):255–265, 1981.

[76] P. A. Bernstein and N. Goodman. The power of natural semijoin. *SIAM J. Comp.*, 10(4):751–771, 1981.

[77] E. Bertino and P. Foscoli. An analytical cost model of object-oriented query costs. In *Proc. Persistent Object Systems*, pages 151–160, 1992.

[78] E. Bertino and D. Musto. Query optimization by using knowledge about data semantics. *Data & Knowledge Engineering*, 9(2):121–155, 1992.

[79] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Trans. on Knowledge and Data Eng.*, 4(3):223–237, June 1992.

[80] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 304–315, 1995.

[81] G. Bhargava, P. Goel, and B. Iyer. Efficient processing of outer joins and aggregate functions. In *Proc. IEEE Conference on Data Engineering*, pages 441–449, 1996.

[82] A. Biliris. An efficient database storage structure for large dynamic objects. In *Proc. IEEE Conference on Data Engineering*, pages 301–308, 1992.

[83] D. Bitton and D. DeWitt. Duplicate record elimination in large data files. *ACM Trans. on Database Systems*, 8(2):255–265, 1983.

[84] J. Blakeley and N. Martin. Join index, materialized view, and hybrid hash-join: a performance analysis. In *Proc. IEEE Conference on Data Engineering*, pages 256–236, 1990.

[85] J. Blakeley, W. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 287–295, 1993.

[86] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 61–71, Washington, D.C., 1986.

[87] B. Blohsfeld, D. Korus, and B. Seeger. A comparison of selectivity estimators for range queries on metric attributes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 239–250, 1999.

[88] P. Bodorik and J.S. Riordon. Distributed query processing optimization objectives. In *Proc. IEEE Conference on Data Engineering*, pages 320–329, 1988.

[89] T. Böhme and E. Rahm. Xmach-1: A benchmark for XML data management. In *BTW*, pages 264–273, 2001.

[90] A. Bolour. Optimal retrieval for small range queries. *SIAM J. of Comput.*, 10(4):721–741, 1981.

[91] P. Bonatti. On the decidability of containment of recursive datalog queries - preliminary report. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 297–306, 2004.

[92] P. Boncz, A. Wilschut, and M. Kersten. Flattening an object algebra to provide performance. In *Proc. IEEE Conference on Data Engineering*, pages 568–577, 1998.

[93] B. Bouchou, M. Halfeld, and F. Alves. Updates and incremental validation of xml documents. In *Int. Workshop on Database Programming Languages*, pages 216–232, 2003.

[94] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged algebraic XPath processing in Natix. In *Proc. IEEE Conference on Data Engineering*, pages 705–716, 2005.

[95] K. Bratbergsengen and K. Norvag. Improved and optimized partitioning techniques in database query procesing. In *Advances in Databases, 15th British National Conference on Databases*, pages 69–83, 1997.

[96] Y. Breitbart and A. Reiter. Algorithms for fast evaluation of boolean expressions. *Acta Informatica*, 4:107–116, 1975.

[97] S. Bressan, M. Lee, Y. Li, Z. Lacroix, and U. Nambiar. The XOO7 XML Management System Benchmark. Technical Report TR21/00, National University of Singapore, 2001.

[98] A. Broder, M. Charikar, and A. Frieze. Minwise independent permutations. In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 327–336, 1998.

[99] K. Brown, M. Carey, and M. Livny. Goal-oriented buffer management revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 353–364, Montreal, Canada, Jun 1996.

[100] N. Bruno and S. Chaudhuri. Exploiting statistics on intermediate tables for query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages ?–?, 2002.

[101] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a multidimensional workload-aware histogram. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 211–222, 2001.

[102] F. Bry. Logical rewritings for improving the evaluation of quantified queries. In *2nd. Symp. on Mathematical Fundamentals of Database Systems*, pages 100–116, June 1989, Visegrad, Hungary, 1989.

[103] F. Bry. Towards an efficient evaluation of general queries: Quantifiers and disjunction processing revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 193–204, 1989.

[104] P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for XML. In *WWW Conference*, pages 201–210, 2001.

[105] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[106] L. Cabibbo and R. Torlone. A framework for the investigation of aggregate functions in database queries. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 383–397, 1999.

[107] J.-Y. Cai, V. Chakaravarthy, R. Kaushik, and J. Naughton. On the complexity of join predicates. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 207–214, 2001.

[108] D. Calvanese, G. DeGiacomo, M. Lenzerini, and M. Vardi. View-based query answering and query containment over semistructured data. In *Int. Workshop on Database Programming Languages*, pages 40–61, 2001.

[109] B. Cao and A. Badia. A nested relational approach to processing sql subqueries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 191–202, 2005.

[110] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *JOURNAL of the ACM*, Computing Surveys:471–522, 1985.

[111] A. F. Cardenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5):253–263, 1975.

[112] M. Carey, D. DeWitt, J. Richardson, and E. Shikita. Object and file management in the EXODUS extensible database system. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–100, 1986.

[113] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, 1997.

[114] M. Carey and D. Kossmann. Processing top N and bottom N queries. *IEEE Data Engineering Bulletin*, 20(3):12–19, 1997.

[115] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 158–169, 1998.

[116] J. Carlis. HAS: A relational algebra operator, or devide is not to conquer. In *Proc. IEEE Conference on Data Engineering*, pages 254–261, 1986.

[117] L. Carlitz, D. Roselle, and R. Scoville. Some remarks on ballot-type sequences of positive integers. *Journal of Combinatorial Theory*, 11:258–271, 1971.

[118] C. R. Carlson and R. S. Kaplan. A generalized access path model and its application to a relational database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 143–154, 1976.

[119] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries. *IEEE Trans. on Software Eng.*, 11(4):324–345, Apr 1985.

[120] S. Ceri and G. Pelagatti. Correctness of query execution strategies in distributed databases. *ACM Trans. on Database Systems*, 8(4):577–607, Dec. 1983.

[121] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill, 1985.

[122] S. Chakravarthy. Devide and conquer: a basis for augmenting a conventional query optimizer with multiple query processing capabilities. In *Proc. IEEE Conference on Data Engineering*, 1991.

[123] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. on Database Systems*, 15(2):162–207, 1990.

[124] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages ?–?, 1986.

[125] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[126] A. Chan and B. Niamir. On estimating cost of accessing records in blocked database organizations. *Comput. J.*, 25(3):368–374, 1982.

[127] C. Chan and B. Ooi. Efficient scheduling of page accesses in index-based join processing. *IEEE Trans. on Knowledge and Data Eng.*, 9(6):1005–1011, 1997.

[128] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. ACM SIGACT Symp. on the Theory of Computing*, pages 77–90, 1977.

[129] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.

[130] S. Chatterji, S. Evani, S. Ganguly, and M. Yemmanuru. On the complexity of approximate query optimization. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 282–292, 2002.

[131] D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *Proc. IEEE Conference on Data Engineering*, pages 524–533, 2001.

[132] D. Chatziantoniou and K. Ross. Querying multiple features in relational databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 295–306, 1996.

[133] D. Chatziantoniou and K. Ross. Groupwise processing of relational queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 476–485, 1997.

[134] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 361–372, 2003.

[135] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proc. IEEE Conference on Data Engineering*, pages 227–238, 2004.

[136] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 397–410, 1999.

[137] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. IEEE Conference on Data Engineering*, pages 190–200, 1995.

[138] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. *Optimizing Queries with Materialized Views*, pages 77–92. MIT Press, 1999.

[139] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 436–447, Seattle, WA, 1998.

[140] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of long running sql queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 803–814, 2004.

[141] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 529–542, 1993.

[142] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 354–366, 1994.

[143] S. Chaudhuri and K. Shim. The promise of early aggregation. Technical report, HP Lab, 1994. Never Appeared.

[144] S. Chaudhuri and K. Shim. An overview of cost-based optimization of queries with aggregates. *IEEE Data Engineering Bulletin*, 18(3):3–9, Sept 1995.

[145] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 87–98, 1996.

[146] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 167–182, 1996.

[147] S. Chaudhuri and M. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 55–66, 1992.

[148] S. Chaudhuri and M. Vardi. Optimization of real conjunctive queries. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 59–70, 1993.

[149] S. Chaudhuri and M. Vardi. Optimization of real conjunctive queries. Technical Report HPL-93-26, HP Software Technology Laboratory, 1993.

[150] S. Chaudhuri and M. Vardi. On the complexity of equivalence between recursive and nonrecursive datalog programs. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 55–66, 1994.

[151] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the *really* hard problems are. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 331–337, 1991.

[152] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 56–70, 1997.

[153] C. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 161–172, 1994.

[154] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 323–336, 1994.

[155] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–282, 2001.

[156] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava. Counting twig matches in a tree. In *Proc. IEEE Conference on Data Engineering*, pages 595–604, 2001.

[157] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity estimation for boolean queries. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 216–225, 2000.

[158] Z. Chen and V. Narasayya. Efficient computation of multiple group by queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 263–274, 2005.

[159] J. Cheng, D. Haderle, R. Hedges, B. Iyer, T. Messenger, C. Mohan, and Y. Wang. An efficient hybrid join algorithm: A DB2 prototype. In *Proc. IEEE Conference on Data Engineering*, pages 171–180, 1991.

[160] J. Cheng, C. Loosley, A. Shibamiya, and P. Worthington. IBM DB2 Performance: design, implementation, and tuning. *IBM Sys. J.*, 23(2):189–210, 1984.

[161] Q. Cheng, J. Gryz, F. Koo, T. Y. Cliff Leung, L. Liu, X. Quian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 687–698, 1999.

[162] M. Cherniack and S. Zdonik. Rule languages and internal algebras for rule-based optimizers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 401–412, 1996.

[163] M. Cherniack and S. Zdonik. Changing the rules: Transformations for rule-based optimizers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 61–72, Seattle, WA, 1998.

[164] T.-Y. Cheung. Estimating block accesses and number of records in file management. *Communications of the ACM*, 25(7):484–487, 1982.

[165] D. M. Chiu and Y. C. Ho. A methodology for interpreting tree queries into optimal semi-join expressions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 169–178, 1980.

[166] H.-T. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 127–141, 1985.

[167] S. Christodoulakis. Estimating selectivities in databases. Tech. Report CSRG-136, University of Toronto, 1981.

[168] S. Christodoulakis. Estimating block transfers and join sizes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 40–54, 1983.

[169] S. Christodoulakis. Estimating block transfers and join sizes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 40–54, 1983.

[170] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

[171] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 8(2):105–115, 1983.

[172] S. Christodoulakis. Estimating block selectivities. *Information Systems*, 9(1):69–79, 1984.

[173] S. Christodoulakis. Estimating record selectivities. *Information Systems*, 9(1):69–69, 1984.

[174] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 9(2):163–186, June 1984.

[175] S. Christodoulakis. Analysis of retrieval performance for records and objects using optical disk technology. *ACM Trans. on Database Systems*, 12(2):137–169, 1987.

[176] V. Christophides, S. Cluet, and G. Moerkotte. Evaluating queries with generalized path expressions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 413–422, 1996.

[177] P.-C. Chu. A contingency approach to estimating record selectivities. *IEEE Trans. on Software Eng.*, 17(6):544–552, 1991.

[178] P.-C. Chu. Estimating block selectivities for physical database design. *IEEE Trans. on Knowledge and Data Eng.*, 4(1):89–98, 1992.

[179] C. Clarke, G. Cormack, and F. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[180] J. Claussen, A. Kemper, and D. Kossmann. Order-preserving hash joins: Sorting (almost) for free. Technical Report MIP-9810, University of Passau, 1998.

[181] J. Claussen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 286–295, 1997.

[182] J. Claussen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. Technical Report MIP–9706, University of Passau, Fak. f. Mathematik u. Informatik, Mar 1997.

[183] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. on Knowledge and Data Eng.*, 12(2):238–260, 2000.

[184] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, 1992.

[185] S. Cluet, C. Delobel, C. Lecluse, and P. Richard. Reloop, an algebra based query language for an object-oriented database system. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1989.

[186] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, pages 226–242, 1993.

[187] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[188] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. Technical Report 95-5, RWTH-Aachen, 1995.

[189] S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. In *Proc. Int. Workshop on Database Programming Languages*, 1995.

[190] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 54–67, 1995.

[191] S. Cluet and G. Moerkotte. Query optimization techniques exploiting class hierarchies. Technical Report 95-7, RWTH-Aachen, 1995.

[192] E. Codd. *Database Systems - Courant Computer Science Symposium*. Prentice Hall, 1972.

[193] E. F. Codd. A database sublanguage founded on the relational calculus. In *Proc. ACM-SIGFIDET Workshop, Datadescription, Access, and Control*, pages 35–68, San Diego, Calif., 1971. ACM.

[194] E. F. Codd. Relational completeness of data base sublanguages. In *Courant Computer Science Symposia No. 6: Data Base Systems*, pages 67–101, New York, 1972. Prentice Hall.

[195] E. F. Codd. Extending the relational database model to capture more meaning. *ACM Trans. on Database Systems*, 4(4):397–434, Dec. 1979.

[196] E. Cohen. Size-estimation framework with applications to transitive closure. *Journal of Comput. Syst. Sciences*, 55:441–453, 1997.

[197] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proc. IEEE Conference on Data Engineering*, pages 489–499, 2000.

[198] S. Cohen, W. Nutt, and Y. Sagiv. Equivalences among aggregate queries with negation. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 215–226, 2001.

[199] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries (extended version). Technical report, Hebrew University of Jerusalem, 2002. available at www.cs.huji.ac.il/ sarina/papers/agg-containment-long.ps.

[200] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 111–125, 2003.

[201] L. Colby. A recursive algebra and query optimization for nested relational algebra. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 273–283, 1989.

[202] L. Colby, A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Supporting multiple view maintenance policies. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 405–416, 1997.

[203] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 268–279, Austin, TX, 1985.

[204] G. Cormack. Data compression on a database system. *Communications of the ACM*, 28(12):1336–1342, 1985.

[205] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[206] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. 2nd Edition.

[207] D. Cornell and P. Yu. Integration of buffer management and query optimization in relational database environments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 247–255, 1989.

[208] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. National Conference on Artificial Intelligence*, pages 21–27, 1993.

[209] K. Culik, T. Ottmann, and D. Wood. Dense multiway trees. *ACM Trans. on Database Systems*, 6(3):486–512, 1981.

[210] M. Dadashzadeh. An improved division operator for relational algebra. *Information Systems*, 14(5):431–437, 1989.

[211] D. Daniels. Query compilation in a distributed database system. Technical Report RJ 3432, IBM Research Laboratory, San Jose, CA, 1982.

[212] D. Das and D. Batory. Praire: A rule specification framework for query optimizers. In *Proc. IEEE Conference on Data Engineering*, pages 201–210, 1995.

[213] C. J. Date. The outer join. In *Proc. of the Int. Conf. on Databases*, Cambridge, England, 1983.

[214] U. Dayal. Processing queries with quantifiers: A horticultural approach. In *ACM Symp. on Principles of Database Systems*, pages 125–136, 1983.

[215] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[216] U. Dayal, N. Goodman, and R.H. Katz. An extended relational algebra with control over duplicate elimination. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 117–123, 1982.

[217] U. Dayal, F. Manola, A. Buchman, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal. Simplifying complex object: The PROBE approach to modelling and querying them. In *H.J. Schek and G. Schlageter (eds.) Proc. BTW*, pages 17–37, 1987.

[218] U. Dayal and J. Smith. PROBE: A knowledge-oriented database management system. In *Proc. Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, 1985.

[219] G. de Balbine. Note on random permutations. *Mathematics of Computation*, 21:710–712, 1967.

[220] D. DeHaan, P.-A. Larson, and J. Zhou. Stacked index views in microsoft sql server. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 179–190, 2005.

[221] R. Demolombe. Estimation of the number of tuples satisfying a query expressed in predicate calculus language. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 55–63, 1980.

[222] P. Denning. Effects of scheduling on file memory operations. In *Proc. AFIPS*, pages 9–21, 1967.

[223] N. Derrett and M.-C. Shan. Rule-based query optimization in IRIS. Technical report, Hewlard-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA94303, 1990.

[224] B. Desai. Performance of a composite attribute and join index. *IEEE Trans. on Software Eng.*, 15(2):142–152, Feb. 1989.

[225] A. Deshpande, M. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 199–210, 2001.

[226] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *IEEE Data Engineering Bulletin*, 22(3):10–18, 1999.

[227] A. Deutsch, B. Ludäscher, and A. Nash. Rewriting queries using views with access patterns under integrity constraints. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 352–367, 2005.

[228] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath. In *Int. Workshop on Knowledge Representation meets Databases (KRDB)*, 2001.

[229] A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Int. Workshop on Database Programming Languages*, pages 21–39, 2001.

[230] A. Deutsch and V. Tannen. Reformulation of xml queries and constraints. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 225–241, 2003.

[231] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 151–1??, 1984.

[232] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 443–452, 1991.

[233] Y. Diao, M. Altinel, M. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. on Database Systems*, 28(4):367–516, 2003.

[234] G. Diehr and A. Saharia. Estimating block accesses in database organizations. *IEEE Trans. on Knowledge and Data Engineering*, 6(3):497–499, 1994.

[235] P. Dietz. Optimal algorithms for list indexing and subset ranking. In *Workshop on Algorithms and Data Structures (LNCS 382)*, pages 39–46, 1989.

[236] Z. Dimitrijevic, R. Rangaswami, E. Chang, D. Watson, and A. Acharya. Diskbench: User-level disk feature extraction tool. Technical report, University of California, Santa Barbara, 2004.

[237] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for hardware performance monitoring on Linux systems. perform internet search for this or similar tools.

[238] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of Top n queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 411–422, 1999.

[239] J. Donovan. Database system approach to management of decision support. *ACM Trans. on Database Systems*, 1(4):344–368, 1976.

[240] M. Drmota, D. Gardy, and B. Gittenberger. A unified presentation of some urn models. *Algorithmica*, 29:120–147, 2001.

[241] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.

[242] O. Duschka. *Query Planning and Optimization in Information Integration*. PhD thesis, Stanford University, 1997.

[243] O. Duschka and M. Genesereth. Answering recursive queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 109–116, 1997.

[244] O. Duschka and M. Genesereth. Query planning with disjunctive sources. In *AAAI Workshop on AI and Information Integration*, 1998.

[245] O. Duschka and M. Gensereth. Answering recursive queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 109–116, 1997.

[246] O. Duschka and A. Levy. Recursive plans for information gathering. In *Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 1997.

[247] W. Effelsberg and T. Härder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, 1984.

[248] S. Eggers, F. Olken, and A. Shoshani. A compression technique for large statistical data bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 424–434, 1981.

[249] S. Eggers and A. Shoshani. Efficient access of compressed data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 205–211, 1980.

[250] J. F. Egler. A procedure for converting logic table conditions into an efficient sequence of test instructions. *Communications of the ACM*, 6(9):510–514, 1963.

[251] M. Eisner and D. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *Journal of the ACM*, 23(4):619–635, 1976.

[252] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2000. 3rd Edition.

[253] R. Epstein. Techniques for processing of aggregates in relational database systems. ERL/UCB Memo M79/8, University of California, Berkeley, 1979.

[254] D. Barbara et al. The new jersey data reduction report. *IEEE Data Engineering Bulletin*, 20(4):3–45, Dec. 1997.

[255] G. Lohman et al. Optimization of nested queries in a distributed relational database. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1984.

[256] N. Roussopoulos et al. The maryland ADMS project: Views R Us. *IEEE Data Engineering Bulletin*, 18(2), 1995.

[257] P. Schwarz et al. Extensibility in the starburst database system. In *Proc. Int. Workshop on Object-Oriented Database Systems*, 1986.

[258] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 216–226, 1996.

[259] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages ?–?, 2001.

[260] C. Faloutsos and I. Kamel. Relaxing the uniformity and indepencence assumptions using the concept of fractal dimensions. *Journal of Computer and Systems Sciences*, 55(2):229–240, 1997.

[261] C. Farr'e, E. Teniente, and T. Urp'i. Query containment checking as a view updating problem. In *Inf. Conf. on Database and Expert Systems Applications (DEXA)*, pages 310–321, 1998.

[262] C. Farr'e, E. Teniente, and T. Urp'i. The constructive method for query containment checking. In *Inf. Conf. on Database and Expert Systems Applications (DEXA)*, pages 583–593, 1999.

[263] C. Farr'e, E. Teniente, and T. Urp'i. Query containment with negated IDB predicates. In *ABDIS*, pages 583–593, 2003.

[264] C. Farr'e, E. Teniente, and T. Urp'i. Checking query containment with the cqc method. *Data & Knowledge Engineering*, 53:163–223, 2005.

[265] L. Fegaras. Optimizing large OODB queries. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 421–422, 1997.

[266] L. Fegaras. A new heuristic for optimizing large queries. In *DEXA*, pages 726–735, 1998.

[267] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 47–58, 1995.

[268] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. on Database Systems*, 25(4):457–516, 2000.

[269] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. Xquery 1.0: An xml query language. Technical report, W3C, 2003. W3C Working Draft.

[270] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *WebDB 2000*, 2000.

[271] T. Fiebig and G. Moerkotte. Algebraic XML construction in Natix. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, pages 212–221, 2001.

[272] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2002.

[273] S. Finkelstein. Common expression analysis in database applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 235–245, 1982.

[274] D. Florescu. *Espaces de Recherche pour l'Optimisation de Requêtes Objet (Search Spaces for Query Optimization)*. PhD thesis, Université de Paris VI, 1996. in French.

[275] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 139–148, 1998.

[276] P. Fortier. *SQL-3, Implementing the SQL Foundation Standard*. McGraw Hill, 1999.

[277] F. Fotouhi and S. Pramanik. Optimal secondary storage access sequence for performing relational join. *IEEE Trans. on Knowledge and Data Eng.*, 1(3):318–328, 1989.

[278] F. Frasincar, G.-J. Houben, and C. Pau. XAL: An algebra for XML query optimization. In *Australasian Database Conference (ADC)*, 2002.

[279] J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Simeon. StatiX: making XML count. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 181–191, 2002.

[280] J. C. Freytag. A rule-based view of query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 173–180, 1987.

[281] J. C. Freytag and N. Goodman. Translating aggregate queries into iterative programs. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages ?–?, 1986.

[282] J. C. Freytag and N. Goodman. On the translation of relational queries into iterative programs. *ACM Trans. on Database Systems*, 14(1):1–27, 1989.

[283] C. Galindo-Legaria. Outerjoins as disjunctions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 348–358, 1994.

[284] C. Galindo-Legaria. Outerjoins as disjunctions. Technical Report CS-R9404, CWI, Amsterdam, NL, 1994.

[285] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 571–581, 2001.

[286] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Cost distribution of search spaces in query optimization. Technical Report CS-R9432, CWI, Amsterdam, NL, 1994.

[287] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fact, randomized join-order selection — why use transformations? In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 85–95, 1994.

[288] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection — why use transformations? Technical Report CS-R–9416, CWI, Amsterdam, NL, 1994.

[289] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. The impact of catalogs and join algorithms on probabilistic query optimization. Technical Report CS-R9459, CWI, Amsterdam, NL, 1994.

[290] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Uniformly-distributed random generation of join orders. Technical Report CS-R9431, CWI, Amsterdam, NL, 1994.

[291] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Cost distribution of search spaces in query optimization. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 280–293, 1995.

[292] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. on Database Systems*, 22(1):43–73, Marc 1997.

[293] C. Galindo-Legaria, A. Rosenthal, and E. Kortright. Expressions, graphs, and algebraic identities for joins, 1991. working paper.

[294] S. Ganapathy and V. Rajaraman. Information theory applied to the conversion of decision tables to computer programs. *Communications of the ACM*, 16:532–539, 1973.

[295] S. Gandeharizadeh, J. Stone, and R. Zimmermann. Techniques to quantify SCSI-2 disk subsystem specifications for multimedia. Technical Report 95-610, University of Southern California, 1995.

[296] S. Ganguly. On the complexity of finding optimal join order sequence for star queries without cross products. personal correspondance, 2000.

[297] S. Ganguly, P. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–281, 1996.

[298] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–33, 1987.

[299] G. Garani and R. Johnson. Joining nested relations and subrelations. *Information Systems*, 25(4):287–307, 2000.

[300] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 2000.

[301] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. A cost model for clustered object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 323–334, 1995.

[302] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. Cost-based selection of path expression processing algorithms in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 390–401, 1996.

[303] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the query optimizer cost model of IRO-DB, an object-oriented federated database system. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 378–389, 1996.

[304] D. Gardy. Normal limiting distributions for projection and semi-join sizes. *SIAM J. Discrete Mathematics*, 5(2):219–248, 1992.

[305] D. Gardy. Join sizes, urn models and normal limiting distributions. *Theoret. Comp. Sci.*, pages 375–414, 1994.

[306] D. Gardy and G. Louchard. Dynamic analysis of some relational databases parameters. *Theor. Comp. Sci*, 144(1/2):125–159, 1995.

[307] D. Gardy and G. Louchard. Dynamic analysis of the sizes of relations. In *Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 433–444, 1995.

[308] D. Gardy and L. Nemirovski. Urn models and yao's formula. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 100–112, 1999.

[309] D. Gardy and C. Puech. On the effect of join operations on relation sizes. *ACM Trans. on Database Systems*, 14(4):574–603, 1989.

[310] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1978.

[311] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[312] M. Garofalakis and P. Gibbons. Wavelet synopses with error guarantees. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 476–487, 2002.

[313] P. Gassner, G. Lohman, and K. Schiefer. Query optimization in the IBM DB2 family. *IEEE Data Engineering Bulletin*, 16:4–18, Dec. 1993.

[314] P. Gassner, G. Lohman, K. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. Technical report rj 9734, IBM, 1994.

[315] A. Van Gelder. Multiple join size estimation by virtual domains. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, 1993.

[316] E. Gelenbe and D. Gardy. On the size of projections: I. *Information Processing Letters*, 14:1, 1982.

[317] E. Gelenbe and D. Gardy. The size of projections of relations satisfying a functional dependency. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 325–333, 1982.

[318] E. Gelenbe and D. Gardy. On the sizes of projections: a generating function approach. *Information Systems*, 9(3/4):231–235, 1984.

[319] E. Gelenbe and D. Gardy. Relational algebra operations and sizes of relations. In *Int. Col. Automata, Languages and Programming (ICALP)*, pages 174–186, 1984.

[320] I. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proc. National Conference on Artificial Intelligence*, pages 28–33, 1993.

[321] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 331–342, 1998.

[322] P. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 466–475, 1997.

[323] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 582–592, 2001.

[324] P. Goel and B. Iyer. SLQ query optimization: reordering for a general class of queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 47–56, 1996.

[325] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[326] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 331–342, 2001.

[327] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. IEEE Conference on Data Engineering*, 1998. to appear.

[328] G. Gorry and S. Morton. A framework for management information systems. *Sloan Management Review*, 13(1):55–70, 1971.

[329] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 95–106, 2002.

[330] G. Gottlob, C. Koch, and R. Pichler. XPath processing in a nutshell. *SIGMOD Record*, 2003.

[331] G. Gottlob, C. Koch, and R. Pichler. Xpath query evaluation: Improving time and space efficiency. In *Proc. IEEE Conference on Data Engineering*, page to appear, 2003.

[332] M. Gouda and U. Dayal. Optimal semijoin schedules for query processing in local distributed database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 164–175, 1981.

[333] P. Goyal. Coding methods for text string search on compressed databases. *Information Systems*, 8(3):231–233, 1983.

[334] G. Graefe. Software modularization with the exodus optimizer generator. *IEEE Database Engineering*, 9(4):37–45, 1986.

[335] G. Graefe. Relational division: Four algorithms and their performance. In *Proc. IEEE Conference on Data Engineering*, pages 94–101, 1989.

[336] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages ?–?, 1990.

[337] G. Graefe. Heap-filter merge join: A new algorithm for joining medium-size inputs. *IEEE Trans. on Software Eng.*, 17(9):979–982, 1991.

[338] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.

[339] G. Graefe. Query evaluation techniques for large databases. Shortened version: [338], July 1993.

[340] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. IEEE Conference on Data Engineering*, pages 406–417, 1994.

[341] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, Sept 1995.

[342] G. Graefe. Executing nested queries. In *BTW*, pages 58–77, 2003.

[343] G. Graefe and R. Cole. Fast algorithms for universal quantification in large databases. Internal report, Portland State University and University of Colorado at Boulder, 19??

[344] G. Graefe and R. Cole. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages ?–?, 1994.

[345] G. Graefe and R. Cole. Fast algorithms for universal quantification in large databases. *ACM Trans. on Database Systems*, ?(?):?–?, ? 1995?

[346] G. Graefe, R. Cole, D. Davison, W. McKenna, and R. Wolniewicz. Extensible query optimization and parallel execution in Volcano. In *Dagstuhl Query Processing Workshop*, pages 337–380, 1991.

[347] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, 1987.

[348] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Trans. on Knowledge and Data Eng.*, 6(6):934–944, Dec. 1994.

[349] G. Graefe and W. McKenna. The Volcano optimizer generator. Tech. Report 563, University of Colorado, Boulder, 1991.

[350] G. Graefe and W. McKenna. Extensibility and search efficiency in the volcano optimizer generator. In *Proc. IEEE Conference on Data Engineering*, pages 209–218, 1993.

[351] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 358–366, 1989.

[352] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2002.

[353] G. Grahne and A. Thomo. Algebraic rewritings for optimizing regular path queries. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 301–315, 2001.

[354] G. Grahne and A. Thomo. New rewritings and optimizations for regular path queries. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 242–258, 2003.

[355] J. Grant, J. Gryz, J. Minker, and L. Raschid. Semantic query optimization for object databases. In *Proc. IEEE Conference on Data Engineering*, pages 444–453, 1997.

[356] J. Gray, editor. *The Benchmark Handbook*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[357] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A relational aggregation operator generlizing group-by, cross-tab, and sub-totals. In *Proc. IEEE Conference on Data Engineering*, pages 152–169, 1996.

[358] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[359] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disk accesses and the 10 byte rule for trading memory for CPU time. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 395–398, 1987.

[360] P. Grefen and R. de By. A multi-set extended relational algebra – a formal approach to a practical issue. In *Proc. IEEE Conference on Data Engineering*, pages 80–88, 1994.

[361] T. Grust. Accelerating XPath location steps. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 109–120, 2002.

[362] T. Grust and M. Van Keulen. Tree awareness for relational database kernels: Staircase join. In *Intelligent Search on XML Data*, pages 231–245, 2003.

[363] T. Grust, M. Van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 524–525, 2003.

[364] T. Grust, M. Van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. on Database Systems*, 29(1):91–131, 2004.

[365] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and application of check constraints in DB2. In *Proc. IEEE Conference on Data Engineering*, pages 551–556, 2001.

[366] E. Gudes and A. Reiter. On evaluating boolean expression. *Software Practice and Experience*, 3:345–350, 1973.

[367] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 471–475, 2001.

[368] H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *Proc. IEEE Conference on Data Engineering*, pages 336–344, 1991.

[369] L. Guo, K. Beyer, J. Shanmugasundaram, and E. Shekita. Efficient inverted lists and query algorithms for structured value ranking in update-intense relational databases. In *Proc. IEEE Conference on Data Engineering*, pages 298–309, 2005.

[370] M. Guo, S. Y. W. Su, and H. Lam. An association algebra for processing object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages ?–?, 1991.

[371] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 358–369, 1995.

[372] A. Gupta, V. Harinarayan, and D. Quass. Generalized projections: A powerful approach to aggregation. Technical Report, 1995.

[373] A. Gupta and I. Mumick. Maintenance of materialized views: problems, techniques and applications. *IEEE Data Engineering Bulletin*, 18(2):3–19, 1995.

[374] R. Güting, R. Zicari, and D. Choy. An algebra for structured office documents. *ACM Trans. on Information Systems*, 7(4):123–157, 1989.

[375] R. H. Güting. Geo-relational algebra: A model and query language for geometric database systems. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proc. of the Intl. Conf. on Extending Database Technology*, pages 506–527, Venice, Italy, Mar 1988. Springer-Verlag, Lecture Notes in Computer Science No. 303.

[376] R. H. Güting. Second-order signature: A tool for specifying data models, query processing, and optimization. Informatik-Bericht 12/1992, ETH Zürich, 1992.

[377] L. Haas, M. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *The VLDB Journal*, 6(3):241–256, 1997.

[378] L. Haas, W. Chang, G. Lohman, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starbust mid-flight: As the dust clears. *IEEE Trans. on Knowledge and Data Eng.*, 2(1):143–160, 1990.

[379] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 377–388, 1989.

[380] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 311–322, 1995.

[381] P. Haas, J. Naughton, S. Seshadri, and A. Swami. Fixed-precision estimation of join selectivity. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 190–?, 1993.

[382] P. Haas and A. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *Proc. IEEE Conference on Data Engineering*, pages 522–531, 1995.

[383] P. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 341–350, 1992.

[384] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec. 2001.

[385] P. A. V. Hall. Common subexpression identification in general algebraic systems. Tech. rep. uksc 0060, IBM UK Scientific Center, Peterlee, England, 1974.

[386] P. A. V. Hall. Optimization of single expressions in a relational database system. *IBM J. Res. Devel.*, 20(3):244–257, 1976.

[387] P. A. V. Hall and S. Todd. Factorization of algebraic expressions. Tech. Report UKSC 0055, IBM UK Scientific Center, Peterlee, England, 1974.

[388] C. Hamalainen. Complexity of query optimisation and evaluation. Master's thesis, Griffith University, Queensland, Australia, 2002.

[389] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 93–101, 1979.

[390] J. Han. Smallest-first query evaluation for database systems. In *Australian Database Conference*, pages ?–?, Christchurch, New Zealand, Jan. 1994.

[391] M. Z. Hanani. An optimal evaluation of boolean expressions in an online query system. *Communications of the ACM*, 20(5):344–347, 1977.

[392] E. Hanson. A performance analysis of view materialization strategies. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 440–453, 1987.

[393] E. Hanson. Processing queries against database procedures. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages ?–?, 1988.

[394] E.N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–?, 1990.

[395] T. Härder. Implementing a generalized access path structure for a relational database system. *ACM Trans. on Database Systems*, 3(3):285–298, 1978.

[396] T. Härder, B. Mitschang, and H. Schöning. Query processing for complex objects. *Data and Knowledge Engineering*, 7(3):181–200, 1992.

[397] T. Härder and E. Rahm. *Datenbanksysteme*. Springer, 1999.

[398] T. Härder, H. Schöning, and A. Sikeler. Parallelism in processing queries on complex objects. In *International Symposium on Databases in Parallel and Distributed Systems*, Ausgin, TX, August 1988.

[399] V. Harinarayan. Issues in interactive aggregation. *IEEE Data Engineering Bulletin*, 20(1):12–18, 1997.

[400] V. Harinarayan and A. Gupta. Generalized projections: a powerful query optimization technique. Technical Report STAN-CS-TN-94-14, Stanford University, 1994.

[401] E. Harris and K. Ramamohanarao. Join algorithm costs revisited. *The VLDB Journal*, 5(1):?–?, Jan 1996.

[402] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Research Report RJ6367, IBM, 1988.

[403] Heller. Rabbit: A performance counter library for Intel/AMD processors and Linux. perform internet search for this or similar tools.

[404] J. Hellerstein. Practical predicate placement. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–335, 1994.

[405] J. Hellerstein and J. Naughton. Query execution techniques for caching expensive methods. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 423–434, 1996.

[406] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 267–277, 1993.

[407] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of xpath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, 2002. 215-224.

[408] S. Helmer, B. König-Ries, and G. Moerkotte. The relational difference calculus and applications. Technical report, Universität Karlsruhe, 1993. (unpublished manuscript).

[409] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. Technical Report 13/96, University of Mannheim, Mannheim, Germany, 1996.

[410] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 386–395, 1997.

[411] S. Helmer and G. Moerkotte. Index structures for databases containing data items with set-valued attributes. Technical Report 2/97, University of Mannheim, 1997.

[412] S. Helmer and G. Moerkotte. A study of four index structures for set-valued attributes of low cardinality. Technical Report 02/99, University of Mannheim, 1999.

[413] S. Helmer and G. Moerkotte. Compiling away set containment and intersection joins. Technical Report 4, University of Mannheim, 2002.

[414] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB Journal*, 12(3):244–261, 2003.

[415] S. Helmer, T. Neumann, and G. Moerkotte. Early grouping gets the skew. Technical Report 9, University of Mannheim, 2002.

[416] S. Helmer, T. Neumann, and G. Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 656–667, 2003.

[417] A. Heuer and M. H. Scholl. Principles of object-oriented query languages. In *Proc. der GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft (BTW)*. Springer, 1991.

[418] J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in xpath. In *Int. Workshop on Database Programming Languages*, pages 54–70, 2003.

[419] D. Hirschberg. On the complexity of searching a set of vectors. *SIAM J. Computing*, 9(1):126–129, 1980.

[420] T. Hogg and C. Williams. Solving the really hard problems with cooperative search. In *Proc. National Conference on Artificial Intelligence*, pages 231–236, 1993.

[421] L. Hong-Cheu and K. Ramamohanarao. Algebraic equivalences among nested relational expressions. In *CIKM*, pages 234–243, 1994.

[422] W.-C. Hou and G. Ozsoyoglu. Statistical estimators for aggregate relational queries. *ACM Trans. on Database Systems*, 16(4):600–654, 1991.

[423] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages ?–?, 2001.

[424] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 26–35, 1997.

[425] F. Hwang and G. Chang. Enumerating consecutive and nested partitions for graphs. Technical Report DIMACS Technical Report 93-15, Rutgers University, 1993.

[426] H.-Y. Hwang and Y.-T. Yu. An analytical method for estimating and interpreting query time. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 347–358, 1987.

[427] L. Hyafil and R. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.

[428] T. Ibaraki and T. Kameda. Optimal nesting for computing n-relational joins. *ACM Trans. on Database Systems*, 9(3):482–502, 1984.

[429] O. Ibarra and J. Su. On the containment and equivalence of database queries with linear constraints. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 32–43, 1997.

[430] A. IJbema and H. Blanken. Estimating bucket accesses: A practical approach. In *Proc. IEEE Conference on Data Engineering*, pages 30–37, 1986.

[431] I. Ilyas, J. Rao, G. Lohman, D. Gao, and E. Lin. Estimating compilation time of a query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 373–384, 2003.

[432] Y. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[433] Y. Ioannidis. *A. Tucker (ed.): The Computer Science and Engineering Handbook*, chapter Query Optimization, pages 1038–1057. CRC Press, 1997.

[434] Y. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. on Database Systems*, 18(4):709–748, 1993.

[435] Y. Ioannidis and V. Poosola. Histogram-based solutions to diverse database estimation problems. *IEEE Data Engineering Bulletin*, 18(3):10–18, Sept 1995.

[436] Y. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations and sets. *ACM Trans. on Database Systems*, 20(3):288–324, 1992.

[437] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 268–277, 1991.

[438] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 312–321, 1990.

[439] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 168–177, 1991.

[440] Y. E. Ioannidis, Y. C. Kang, and T. Zhang. Cost wells in random graphs. personal communication, Dec. 1996.

[441] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. Tech. report, University of Wisconsin, Madison, 1992.

[442] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 103–114, 1992.

[443] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 9–22, 1987.

[444] D. Jacobsen and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Hewlett-Packard Laboratories, 1991.

[445] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *VLDB Journal*, 2003. to appear.

[446] H. V. Jagadish, H. Jin, B. C. Ooi, and K.-L. Tan. Global optimization of histograms. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 223–234, 2001.

[447] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multidimensional substring selectivity estimation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 387–398, 1999.

[448] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 275–286, 1998.

[449] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. Int. Workshop on Database Programming Languages*, pages 149–164, 2001.

[450] H. V. Jagadish, T. R. Ng, and D. Srivastava. Substring selectivity estimation. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 249–260, 1999.

[451] R. Jain and I. Chlamtac. The p^2 algorithm for dynamic calculation for quantiles and histograms without storing observations. *Communications of the ACM*, 26(10):1076–1085, 1985.

[452] C. Janssen. The visual profiler. perform internet search for this or similar tools.

[453] M. Jarke. Common subexpression isolation in multiple query optimization. In W. Kim, D. Reiner, and D. Batory, editors, *Topics in Information Systems. Query Processing in Database Systems*, pages 191–205, 1985.

[454] M. Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems, W. Kim, D. Reiner, D. Batory (Eds.)*, pages 191–205, 1985.

[455] M. Jarke and J.Koch. Range nesting: A fast method to evaluate quantified queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 196–206, 1983.

[456] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, pages 111–152, Jun 1984.

[457] P. Jenq, D. Woelk, W. Kim, and W. Lee. Query processing in distributed ORION. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 169–187, 1990.

[458] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 88–99, 1988.

[459] D. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 164–168, 1982.

[460] D. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comp. Sys. Sci.*, 28(1):167–189, 1984.

[461] D. S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM J. Comput.*, 12(4):616–640, 1983.

[462] B. Jonsson, M. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 118–129, 1998.

[463] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 106–117, 1998.

[464] J. Kahn, G. Kalai, and N. Linial. The influence of variables on boolean functions. In *IEEE ???*, pages 68–80, 1988.

[465] M. Kamath and K. Ramamritham. Bucket skip merge join: A scalable algorithm for join processing in very large databases using indexes. Technical Report 20, University of Massachusetts at Amherst, Amherst, MA, 1996.

[466] Y. Kambayashi. Processing cyclic queries. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 62–78, 1985.

[467] Y. Kambayashi and M. Yoshikawa. Query processing utilizing dependencies and horizontal decomposition. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 55–68, 1983.

[468] R. Kaushik, R. Ramakrishnan, and V. Chakaravarthy. Synopses for query optimization: A space-complete perspective. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 201–209, 2004.

[469] A. Kawaguchi, D. Lieuwen, I. Mumick, and K. Ross. Implementing incremental view maintenance in nested data models. In *Proc. Int. Workshop on Database Programming Languages*, 1997.

[470] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *PDIS*, pages 229–238, 1994.

[471] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 148–157, 1991.

[472] A. Kemper and A. Eickler. *Datenbanksysteme*. Oldenbourg, 2001. 4th Edition.

[473] A. Kemper and G. Moerkotte. Advanced query processing in object bases: A comprehensive approach to access support, query transformation and evaluation. Technical Report 27/90, University of Karlsruhe, 1990.

[474] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 294–305, 1990.

[475] A. Kemper and G. Moerkotte. Query optimization in object bases: Exploiting relational techniques. In *Proc. Dagstuhl Workshop on Query Optimization (J.-C. Freytag, D. Maier und G. Vossen (eds.))*. Morgan-Kaufman, 1993.

[476] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 543–554, 1993.

[477] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 336–347, 1994.

[478] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimierung Boolescher Ausdrücke in Objektbanken. In *Grundlagen von Datenbanken (Eds. U. Lipeck, R. Manthey)*, pages 91–95, 1992.

[479] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimization of boolean expressions in object bases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 79–90, 1992.

[480] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[481] W. Kiessling. On semantic reefs and efficient processing of correlation queries with aggregates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 241–250, 1985.

[482] K. C. Kim, W. Kim, D. Woelk, and A. Dale. Acyclic query processing in object-oriented databases. In *Proc. of the Entity Relationship Conf.*, 1988.

[483] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sep 82.

[484] J. J. King. Exploring the use of domain knowledge for query processing efficiency. Technical Report STAN-CS-79-781, Computer Science Department, Stanford University, 1979.

[485] J. J. King. Quist: A system for semantik query optimization in relational databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 510–517, 1981.

[486] A. Klausner. *Multirelations in Relational Databases*. PhD thesis, Harvard University, Cambridge, 1986.

[487] M. Klettke, L. Schneider, and A. Heuer. Metrics for XML Document Collections. In *EDBT Workshop XML-Based Data Management (XMLDM)*, pages 15–28, 2002.

[488] A. Klug. Calculating constraints on relational expressions. *ACM Trans. on Database Systems*, 5(3):260–290, 1980.

[489] A. Klug. Access paths in the "ABE" statistical query facility. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 161–173, 1982.

[490] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[491] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988. Written 1982 and published posthumously.

[492] D. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1997.

[493] J. Koch. *Relationale Anfragen: Zerlegung und Optimierung*. Informatik-Fachberichte 101. Springer-Verlag, 1985.

[494] J. Kollias. An estimate for seek time for batched searching of random or index sequential structured files. *The Computer Journal*, 21(2):132–133, 1978.

[495] A. König and G. Weikum. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 423–434, 1999.

[496] B. König-Ries, S. Helmer, and G. Moerkotte. An experimental study on the complexity of left-deep join ordering problems for cyclic queries. Working Draft, 1994.

[497] B. König-Ries, S. Helmer, and G. Moerkotte. An experimental study on the complexity of left-deep join ordering problems for cyclic queries. Technical Report 95-4, RWTH-Aachen, 1995.

[498] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[499] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25(1):43–82, 2000.

[500] N. Koudas, S. Muthukrishnan, and D. Srivastava. Optimal histograms for hierarchical range queries. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 196–204, 2000.

[501] W. Kowarschick. Semantic optimization: What are disjunctive residues useful for? *SIGMOD Record*, 21(3):26–32, September 1992.

[502] D. Kreher and D. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, 1999.

[503] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 128–137, 1986.

[504] P. Krishnan, J. Vitter, and B. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 282–293, 1996.

[505] A. Kumar and M. Stonebraker. The effect of join selectivities on optimal nesting order. *SIGMOD Record*, 16(1):28–41, 1987.

[506] I. Kunen and D. Suciu. A scalable algorithm for query minimization. ask Dan for more information, year.

[507] S. Kwan and H. Strong. Index path length evaluation for the research storage system of system r. Technical Report RJ2736, IBM Research Laboratory, San Jose, 1980.

[508] L. Lakshman and R. Missaoui. Pushing semantics inside recursion: A general framework for semantic optimization of recursive queries. In *Proc. IEEE Conference on Data Engineering*, pages 211–220, 1995.

[509] S. Lang and Y. Manolopoulos. Efficient expressions for completely and partly unsuccessful batched search of tree-structured files. *IEEE Trans. on Software Eng.*, 16(12):1433–1435, 1990.

[510] S.-D. Lang, J. Driscoll, and J. Jou. A unified analysis of batched searching of sequential and tree-structured files. *ACM Trans. on Database Systems*, 14(4):604–618, 1989.

[511] T. Lang, C. Wood, and I. Fernandez. Database buffer paging in virtual storage systems. *ACM Trans. on Database Systems*, 2(4):339–351, 1977.

[512] R. Lanzelotte and J.-P. Cheiney. Adapting relational optimisation technology for deductive and object-oriented declarative database languages. In *Proc. Int. Workshop on Database Programming Languages*, pages 322–336, 1991.

[513] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 363–373, 1991.

[514] R. Lanzelotte, P. Valduriez, and M. Zait. Optimization of object-oriented recursive queries using cost-controlled stragegies. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 256–265, 1992.

[515] R. Lanzelotte, P. Valduriez, and M. Zäit. On the effectiveness of optimization search strategies for parallel execution. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 493–504, 1993.

[516] R. Lanzelotte, P. Valduriez, M. Ziane, and J.-P. Cheiney. Optimization of nonrecursive queries in OODBMs. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 1–21, 1991.

[517] P.-A. Larson. Data reduction by partial preaggregation. In *Proc. IEEE Conference on Data Engineering*, pages 706–715, 2002.

[518] P.-Å. Larson and H. Yang. Computing queries from derived relations. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 259–269, 1985.

[519] Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*, pages 492–503, 2004.

[520] E. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Ann. Discrete Math.*, 2:75–90, 1978.

[521] C. Lee, C.-S. Shih, and Y.-H. Chen. Optimizing large join queries using a graph-based approach. *IEEE Trans. on Knowledge and Data Eng.*, 13(2):298–315, 2001.

[522] J.-H. Lee, D.-H. Kim, and C.-W Chung. Multi-dimensional selectivity estimation using compressed histogram information. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 205–214, 1999.

[523] M. K. Lee, J. C. Freytag, and G. M. Lohman. Implementing an interpreter for functional rules in a query optimizer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 218–239, 1988.

[524] M. K. Lee, J. C. Freytag, and G. M. Lohman. Implementing an optimizer for functional rules in a query optimizer. Technical Report RJ 6125, IBM Almaden Research Center, San Jose, CA, 1988.

[525] M.K. Lee, J.C, Freytag, and G.M. Lohman. Implementing an interpreter for functional rules in a query optimizer. Research report RJ 6125, IBM, 1988.

[526] T. Lehman and B. Lindsay. The Starburst long field manager. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 375–383, 1989.

[527] K. Lehnert. *Regelbasierte Beschreibung von Optimierungsverfahren für relationale Datenbankanfragesprachen*. PhD thesis, Technische Universität München, 8000 München, West Germany, Dec 1988.

[528] A. Lerner and D. Shasha. AQuery: query language for ordered data, optimization techniques, and experiments. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 345–356, 2003.

[529] H. Leslie, R. Jain, D. Birdsall, and H. Yaghmai. Efficient search of multidimensional B-trees. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 710–719, 1995.

[530] C. Leung, H. Pirahesh, and P. Seshadri. Query rewrite optimization rules in IBM DB2 universal database. Research Report RJ 10103 (91919), IBM Almaden Research Division, January 1998.

[531] M. Levene and G. Loizou. Correction to null values in nested relational databases by m. roth and h. korth and a. silberschatz. *Acta Informatica*, 28(6):603–605, 1991.

[532] M. Levene and G. Loizou. A fully precise null extended nested relational algebra. *Fundamenta Informaticae*, 19(3/4):303–342, 1993.

[533] A. Levy, A. Mendelzon, and Y. Sagiv. Answering queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages ?–?, 1995.

[534] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. *Answering Queries Using Views*, pages 93–106. MIT Press, 1999.

[535] A. Levy, A. Mendelzon, D. Srivastava, and Y. Sagiv. Answering queries using views. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 95–104, 1995.

[536] A. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate movearound. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 96–107, 1994.

[537] A.Y. Levy and I.S. Mumick. Reasoning with aggregation constraints. In P. Apers, M. Bouzeghoub, and G. Gardarin, editors, *Proc. European Conf. on Extending Database Technology (EDBT)*, Lecture Notes in Computer Science, pages 514–534. Springer, March 1996.

[538] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[539] C. Li, K. Chang, I. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 131–142, 2005.

[540] D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comp.*, 11(2):329–343, 1982.

[541] J. Liebehenschel. Ranking and unranking of lexicographically ordered words: An average-case analysis. *J. of Automata, Languages, and Combinatorics*, 2:227–268, 1997.

[542] J. Liebehenschel. Lexicographical generation of a generalized dyck language. Technical Report 5/98, University of Frankfurt, 1998.

[543] J. Liebehenschel. *Lexikographische Generierung, Ranking und Unranking kombinatorisher Objekt: Eine Average-Case Analyse*. PhD thesis, University of Frankfurt, 2000.

[544] H. Liefke. Horizontal query optimization on ordered semistructured data. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.

[545] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. XPathLearner: An on-line self-tuning Markov histogram for XML path selectivity estimation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 442–453, 2002.

[546] J. Lin and M. Ozsoyoglu. Processing OODB queries by O-algebra. In *Int. Conference on Information and Knowledge Management (CIKM)*, pages 134–142, 1996.

[547] Y. Ling and W. Sun. An evaluation of sampling-based size estimation methods for selections in database systems. In *Proc. IEEE Conference on Data Engineering*, pages 532–539, 1995.

[548] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–11, 1990.

[549] S. Listgarten and M.-A. Neimat. Modelling costs for a MM-DBMS. In *Proc. Int. Workshop on Real-Time Databases, Issues and Applications*, pages 72–78, 1996.

[550] J. W. S. Liu. Algorithms for parsing search queries in systems with inverted file organization. *ACM Trans. on Database Systems*, 1(4):299–316, 1976.

[551] M.-L. Lo and C. Ravishankar. Towards eliminating random I/O in hash joins. In *Proc. IEEE Conference on Data Engineering*, pages 422–429, 1996.

[552] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. Research report rj 5992, IBM, 1987.

[553] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, 1988.

[554] D. Lomet. B-tree page size when caching is considered. *ACM SIGMOD Record*, 27(3):28–32, 1998.

[555] R. Lorie. XRM - an extended (N-ary) relational model. Technical Report 320-2096, IBM Cambridge Scientific Center, 1974.

[556] H. Lu and K.-L. Tan. On sort-merge algorithms for band joins. *IEEE Trans. on Knowledge and Data Eng.*, 7(3):508–510, Jun 1995.

[557] W. S. Luk. On estimating block accesses in database organizations. *Communications of the ACM*, 26(11):945–947, 1983.

[558] G. Luo, J. Naughton, C. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 791–802, 2004.

[559] G. Luo, J. Naughton, C. Ellmann, and M. Watzke. Increasing the accuracy and coverage of SQL progress indicators. In *Proc. IEEE Conference on Data Engineering*, pages 853–864, 2005.

[560] C. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distribution of column values. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 240–251, 1988.

[561] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 149–159, 1986.

[562] L. F. Mackert and G. M. Lohman. Index scans using a finite LRU buffer: A validated i/o model. *ACM Trans. on Database Systems*, 14(3):401–425, 1989.

[563] D. Maier and D. S. Warren. Incorporating computed relations in relational databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 176–187, 1981.

[564] A. Makinouchi, M. Tezuka, H. Kitakami, and S. Adachi. The optimization strategy for query evaluation in RDB/V1. In *Proc. IEEE Conference on Data Engineering*, pages 518–529, 1981.

[565] T. Malkemus, S. Padmanabhan, and B. Bhattacharjee. Predicate derivation and monotonicity detection in DB2 UDB. In *Proc. IEEE Conference on Data Engineering*, pages ?–?, 2005.

[566] C. V. Malley and S. B. Zdonik. A knowledge-based approach to query optimization. In *Proc. Int. Conf. on Expert Database Systems*, pages 329–344, 1987.

[567] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 157–168, 2003.

[568] S. Manegold, P. Boncz, and M. Kersten. Generic database cost models for hierarchical memory systems. Technical report, CWI Amsterdam, 2002.

[569] G. Manku, S. Rajagopalan, and B. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 426–435, Seattle, WA, 1998.

[570] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, 1988.

[571] Y. Manolopoulos and J. Kollias. Estimating disk head movement in batched searching. *BIT*, 28:27–36, 1988.

[572] Y. Manolopoulos, J. Kollias, and M. Hatzopoulos. Sequential vs. binary batched search. *The Computer Journal*, 29(4):368–372, 1986.

[573] Y. Manopoulos and J. Kollias. Performance of a two-headed disk system when serving database queries under the scan policy. *ACM Trans. on Database Systems*, 14(3):425–442, 1989.

[574] S. March and D. Severence. The determination of efficient record segmentation and blocking factors for shared data files. *ACM Trans. on Database Systems*, 2(3):279–296, 1977.

[575] R. Marek and E. Rahm. TID hash joins. In *Int. Conference on Information and Knowledge Management (CIKM)*, pages 42–49, 1994.

[576] Y. Matias, J. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 37–48, 1998.

[577] N. May, S. Helmer, C.-C. Kanne, and G. Moerkotte. Xquery processing in natix with an emphasis on join ordering. In *Int. Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, pages 49–54, 2004.

[578] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. Technical report, University of Mannheim, 2003.

[579] N. May, S. Helmer, and G. Moerkotte. Quantifiers in XQuery. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, pages 313–316, 2003.

[580] N. May, S. Helmer, and G. Moerkotte. Three Cases for Query Decorrelation in XQuery. In *Int. XML Database Symp. (XSym)*, pages 70–84, 2003.

[581] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Proc. IEEE Conference on Data Engineering*, pages 239–250, 2004.

[582] J. McHugh and J. Widom. Query optimization for XML. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 315–326, 1999.

[583] N. Megiddo and D. Modha. Outperforming LRU with an adpative replacement cache algorithm. *IEEE Computer*, 37(4):58–65, 2004.

[584] S. Melnik and H. Garcia-Molina. Divide-and-conquer algorithm for computing set containment joins. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 427–444, 2002.

[585] S. Melnik and H. Garcia-Molina. Adaptive algorithms for set containment joins. *ACM Trans. on Database Systems*, 28(1):56–99, 2003.

[586] T. H. Merrett. Database cost analysis: a top down approach. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 135–143, 1977.

[587] T. H. Merrett, Y. Kambayashi, and H. Yasuura. Scheduling of page-fetches in join operations. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 488–498, 1981.

[588] R. Van Meter. Observing the effects of multi-zone disks. In *USENIX Annual Technical Conference*, 1997.

[589] G. Miklau and D. Suciu. Containment and equivalence of xpath expressions. *Journal of the ACM*, 51(1):2–45, 2002.

[590] T. Milo and D. Suciu. Index structures for path expressions. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 277–295, 1999.

[591] M. Minoux. *Mathematical Programming. Theory and Algorithms*. Wiley, 1986.

[592] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proc. National Conference on Artificial Intelligence*, pages 459–465, 1992.

[593] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database*. PhD thesis, Brown University, Providence, RI 02912, 1993.

[594] G. Mitchell, U. Dayal, and S. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages ?–?, 1993.

[595] G. Mitchell, S. Zdonik, and U. Dayal. Object-oriented query optimization: What's the problem? Technical Report CS-91-41, Brown University, 1991.

[596] G. Mitchell, S. Zdonik, and U. Dayal. An architecture for query processing in persistent object stores. In *Proc. of the Hawaiian Conf. on Computer and System Sciences*, pages 787–798, 1992.

[597] G. Mitchell, S. Zdonik, and U. Dayal. *A. Dogac and M. T. Özsu and A. Biliris, and T. Sellis: Object-Oriented Database Systems*, chapter Optimization of Object-Oriented Queries: Problems and Applications, pages 119–146. NATO ASI Series F: Computer and Systems Sciences, Vol. 130. Springer, 1994.

[598] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 476–487, 1998.

[599] G. Moerkotte. Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P. Technical Report 12, University of Mannheim, 2003.

[600] G. Moerkotte. Dp-counter analytics. Technical Report 2, University of Mannheim, 2006.

[601] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy trees without cross products. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages ?–?, 2006.

[602] C. Mohan. Interactions between query optimization and concurrency control. In *Int. Workshop on RIDE*, 1992.

[603] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Int. Conf. on Extended Database Technology (EDBT)*, pages 29–43, 1990.

[604] C. Monma and J. Sidney. Sequencing with series-parallel precedence constraints. *Math. Oper. Res.*, 4:215–224, 1979.

[605] A. I. Montgomery, D. J. D'Souza, and S. B. Lee. The cost of relational algebraic operations in skewed data: estimates and experiments. In *Information Processing*, pages 235–241, New York, 1983. Elsevier North-Holland.

[606] T. Morzy, M. Matyasiak, and S. Salza. Tabu search optimization of large join queries. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 309–322, 1994.

[607] L. Moses and R. Oakland. *Tables of Random Permutations*. Stanford University Press, 1963.

[608] R. Mukkamala and S. Jajodia. A note on estimating the cardinality of the projection of a database relation. *ACM Trans. on Database Systems*, 16(3):564–566, Sept. 1991.

[609] I. Mumick, S. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 247–258, 1990.

[610] I. Mumick and H. Pirahesh. Implementation of magic sets in a relational database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 103–114, 1994.

[611] I. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 264–277, 1990.

[612] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.

[613] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.

[614] M. Muralikrishna and D.J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 28–36, 1988.

[615] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[616] R. Nakano. Translation with optimization from relational calculus to relational algebra having aggregate funktions. *ACM Trans. on Database Systems*, 15(4):518–557, 1990.

[617] T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 960–971, 2004.

[618] T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *Proc. IEEE Conference on Data Engineering*, pages 461–472, 2004.

[619] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 315–329, 2003.

[620] S. Ng. Advances in disk technology: Performance issues. *IEEE Computer*, 31(5):75–81, 1998.

[621] W. Ng and C. Ravishankar. Relational database compression using augmented vector quantization. In *Proc. IEEE Conference on Data Engineering*, pages 540–549, 1995.

[622] S. Nigam and K. Davis. A semantic query optimization algorithm for object-oriented databases. In *Second International Workshop on Constraint Database Systems*, pages 329–344, 1997.

[623] E. Omicienski. Heuristics for join processing using nonclustered indexes. *IEEE Trans. on Software Eng.*, 15(1):18–25, Feb. 1989.

[624] P. O'Neil. *Database Principles, Programming, Performance*. Morgan Kaufmann, 1994.

[625] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 38–49, 1997.

[626] K. Ono and G. Lohman. Extensible enumeration of feasible joins for relational query optimization. Technical Report RJ 6625, IBM Almaden Research Center, 1988.

[627] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 314–325, 1990.

[628] J. A. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 403–412, 1992.

[629] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Eng.*, 14(5):611–629, 1988.

[630] M. Ortega-Binderberger, K. Chakrabarti, and S. Mehrotra. An approach to integrating query refinement in sql. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 15–33, 2002.

[631] S. Osborn. Identity, equality and query optimization. In *Proc. OODB*, 1989.

[632] N. Ott and K. Horlaender. Removing redundant joins in queries involving views. Technical Report TR-82.03.003, IBM Heidelberg Scientific Center, Heidelberg, 1982.

[633] G. Ozsoyoglu, V. Matos, and Z. M. Ozsoyoglu. Query processing techniques in the Summary-Table-by-Example database query language. *ACM Trans. on Database Systems*, 14(4):526–573, 1989.

[634] G. Özsoyoglu, Z. M. Özsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. on Database Systems*, 12(4):566–592, Dec 1987.

[635] G. Ozsoyoglu and H. Wang. A relational calculus with set operators, its safety and equivalent graphical languages. *IEEE Trans. on Software Eng.*, SE-15(9):1038–1052, 1989.

[636] T. Özsu and J. Blakeley. *W. Kim (ed.): Modern Database Systems*, chapter Query Processing in Object-Oriented Database Systems, pages 146–174. Addison Wesley, 1995.

[637] T. Özsu and D. Meechan. Finding heuristics for processing selection queries in relational database systems. *Information Systems*, 15(3):359–373, 1990.

[638] T. Özsu and D. Meechan. Join processing heuristics in relational database systems. *Information Systems*, 15(4):429–444, 1990.

[639] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1999.

[640] T. Özsu and B. Yao. Evaluation of DBMSs using XBench benchmark. Technical Report CS-2003-24, University of Waterloo, 2003.

[641] P. Palvia. Expressions for batched searching of sequential and hierarchical files. *ACM Trans. on Database Systems*, 10(1):97–106, 1985.

[642] P. Palvia and S. March. Approximating block accesses in database organizations. *Information Processing Letters*, 19:75–79, 1984.

[643] S. Papadimitriou, H. Kitagawa, P. Gibbons, and C. Faloutsos. LOCI: Fast outlier detection using local correlation integral. In *ICDE*, pages 315–, 2003.

[644] V. Papadimos and D. Maier. Mutant query plans. *Information & Software Technology*, 44(4):197–206, 2002.

[645] Y. Papakonstantinou and V. Vianu. Incremental validation of xml documents. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 47–63, 2003.

[646] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshops*, pages 128–147, 2002.

[647] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, A. Niermann, and Y. Wu. A physical algebra for XML. Technical report, University of Michigan, 2002.

[648] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. on Database Systems*, 17(1):65–93, Mar 1992.

[649] C.-S. Park, M. Kim, and Y.-J. Lee. Rewriting OLAP queries using materialized views and dimension hierarchies in data. In *Proc. IEEE Conference on Data Engineering*, pages 515–523, 2001.

[650] J. Patel, M. Carey, and M. Vernon. Accurate modeling of the hybrid hash join algorithm. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 56–66, 1994.

[651] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. Technical Report CMU-CS-95-134, Carnegie Mellon University, 1995.

[652] R. Patterson, G. Gibson, and M. Sayanarayanan. A status report on research in transparent informed prefetching. Technical Report CMU-CS-93-113, Carnegie Mellon University, 1993.

[653] G. Paulley and P.-A. Larson. Exploiting uniqueness in query optimization. In *Proc. IEEE Conference on Data Engineering*, pages 68–79, 1994.

[654] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, 1984.

[655] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. Complexity of transformation-based optimizers and duplicate-free generation of alternatives. Technical Report CS-R9639, CWI, 1996.

[656] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 306–315, 1997.

[657] A. Pellenkoft, C. Galindo-Legaria, and M. Kersten. Duplicate-free generation of alternatives in transformation-based optimizers. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 117–124, 1997.

[658] M. Pettersson. Linux x86 performance monitoring counters driver. perform internet search for this or similar tools.

[659] M. Pezarro. A note on estimating hit ratios for direct-access storage devices. *The Computer Journal*, 19(3):271–272, 1976.

[660] B. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 256–276, 1984.

[661] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/rule-based query rewrite optimization in Starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 39–48, 1992.

[662] H. Pirahesh, T. Leung, and W. Hassan. A rule engine for query transformation in Starburst and IBM DB2 C/S DBMS. In *Proc. IEEE Conference on Data Engineering*, pages 391–400, 1997.

[663] A. Pirotte. Fundamental and secondary issues in the design of non-procedural relational languages. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 239–250, 1979.

[664] M. Piwowarski. Comments on batched searching of sequential and tree-structured files. *ACM Trans. on Database Systems*, 10(2):285–287, 1985.

[665] N. Plyzotis and M. Garofalakis. XSKETCH synopsis for XML. In *Hellenic Data Management Symposium 02*, 2002.

[666] S. L. Pollack. Conversion of limited entry decision tables to computer programs. *Communications of the ACM*, 8(11):677–682, 1965.

[667] N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 358–369, 2002.

[668] N. Polyzotis and M. Garofalakis. Structure and value synopsis for XML data graphs. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 466–477, 2002.

[669] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity estimation for XML twigs. In *Proc. IEEE Conference on Data Engineering*, pages 264–275, 2002.

[670] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value indepencence assumption. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 486–495, 1997.

[671] V. Poosola, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimates of range predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 294–305, 1996.

[672] S. Pramanik and D. Ittner. Use of graph-theoretic models for optimal relational database accesses to perform joins. *ACM Trans. on Database Systems*, 10(1):57–74, 1985.

[673] W. Press, S. Teukolsky, and W. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1993.

[674] X. Qian. Query folding. In *Proc. IEEE Conference on Data Engineering*, pages 48–55, 1996.

[675] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 393–404, 1997.

[676] Y.-J. Qyang. A tight upper bound for the lumped disk seek time for the SCAN disk scheduling policy. *Information Processing Letters*, 54:355–358, 1995.

[677] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverwaltung*. Addison-Wesley, 1994.

[678] A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, PODS, 1995.

[679] Bernhard Mitschang Ralf Rantzau, Leonard D. Shapiro and Quan Wang. Algorithms and applications for universal quantification in relational databases. *Information Systems*, 28(1-2):3–32, 2003.

[680] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2000. 2nd Edition.

[681] V. Ramam, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query optimization. In *Proc. IEEE Conference on Data Engineering*, 2003.

[682] K. Ramamohanarao, J. Lloyd, and J. Thom. Partial-match retrieval using hashing descriptors. *ACM Trans. on Database Systems*, 8(4):552–576, 1983.

[683] M. Ramanath, L. Zhang, J. Freire, and J. Haritsa. IMAX: Incremental maintenance of schema-based xXML statistics. In *Proc. IEEE Conference on Data Engineering*, pages 273–284, 2005.

[684] K. Ramasamy, J. Naughton, and D. Maier. High performance implementation techniques for set-valued attributes. Technical report, University of Wisconsin, Wisconsin, 2000.

[685] K. Ramasamy, J. Patel, J. Naughton, and R. Kaushik. Set containment joins: The good, the bad, and the ugly. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 351–362, 2000.

[686] S. Ramaswamy and P. Kanellakis. OODB indexing by class division. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 139–150, 1995.

[687] R. Rantzau, L. Shapiro, B. Mitschang, and Q. Wang. Universal quantification in relational databases: A classification of data and algorithms. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 445–463, 2002.

[688] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen. Using EELs: A practical approach to outerjoin and antijoin reordering. In *Proc. IEEE Conference on Data Engineering*, pages 595–606, 2001.

[689] J. Rao and K. Ross. Reusing invariants: A new strategy for correlated queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 37–48, Seattle, WA, 1998.

[690] S. Rao, A. Badia, and D. Van Gucht. Providing better support for a class of decision support queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 217–227, 1996.

[691] G. Ray, J. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[692] D. Reiner and A. Rosenthal. Strategy spaces and abstract target machines for query optimization. *Database Engineering*, 5(3):56–60, Sept. 1982.

[693] D. Reiner and A. Rosenthal. Querying relational views of networks. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, pages 109–124, 1985.

[694] E. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall, 1977.

[695] L. T. Reinwald and R. M. Soland. Conversion of limited entry decision tables to optimal computer programs I: minimum average processing time. *Journal of the ACM*, 13(3):339–358, 1966.

[696] F. Reiss and T. Kanungo. A characterization of the sensitivity of query optimization to storage access cost parameters. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 385–396, 2003.

[697] A. Reiter, A. Clute, and J. Tenenbaum. Representation and execution of searches over large tree-structured data bases. In *Proc. IFIP Congress, Booklet TA-3*, pages 134–144, 1971.

[698] P. Richard. Evaluation of the size of a query expressed in relational algebra. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 155–163, 1981.

[699] R. Van De Riet, A. Wassermann, M. Kersten, and W. De Jonge. High-level programming features for improving the efficiency of a relational database system. *ACM Trans. on Database Systems*, 6(3):464–485, 1981.

[700] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.

[701] D.J. Rosenkrantz and M.B. Hunt. Processing conjunctive predicates and queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 64–74, 1980.

[702] A. Rosenthal. Note on the expected size of a join. *SIGMOD Record*, 11(4):19–25, 1981.

[703] A. Rosenthal and U. S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 230–239, 1988.

[704] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299, 1990.

[705] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. Traversal recursion: a practical approach to supporting recursive applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 166–167, 1986.

[706] A. Rosenthal and P. Helman. Understanding and extending transformation-based optimizers. *IEEE Data Engineering*, 9(4):44–51, 1986.

[707] A. Rosenthal and D. Reiner. An architecture for query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 246–255, 1982.

[708] A. Rosenthal and D. Reiner. Extending the algebraic framework of query processing to handle outerjoins. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 334–343, 1984.

[709] A. Rosenthal and D. Reiner. Querying relational views of networks. In W. Kim, D. Reiner, and D. Batory, editors, *Query Processing in Database Systems*, New York, 1984. Springer.

[710] A. Rosenthal, C. Rich, and M. Scholl. Reducing duplicate work in relational join(s): a modular approach using nested relations. Technical report, ETH Zürich, 1991.

[711] K. Ross. Conjunctive selection conditions in main memory. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 108–120, 2002.

[712] K. Ross, D. Srivastava, and D. Chatziantoniou. Complex aggregation at multiple granularities. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 263–278, 1998.

[713] M. Roth and S. Horn. Database compression. *SIGMOD Record*, 22(3):31–39, 1993.

[714] M. Roth, H. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 13(4):389–417, 1988. see also [836].

[715] M. Roth, H. Korth, and A. Silberschatz. Null values in nested relational databases. *Acta Informatica*, 26(7):615–642, 1989.

[716] M. Roth, H. Korth, and A. Silberschatz. Addendum to null values in nested relational databases. *Acta Informatica*, 28(6):607–610, 1991.

[717] N. Roussopoulos. View indexing in relational databases. *ACM Trans. on Database Systems*, 7(2):258–290, 1982.

[718] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[719] K. Runapongsa, J. Patel, H. Jagadish, and S. AlKhalifa. The michigan benchmark. Technical report, University of Michigan, 2002.

[720] G. Sacco. Index access with a finite buffer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 301–309, 1887.

[721] G. Sacco and M. Schkolnick. A technique for managing the buffer pool in a relational system using the hot set model. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 257–262, 1982.

[722] G. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, 1986.

[723] G. M. Sacco. Fragmentation: A technique for efficient query processing. *ACM Trans. on Database Systems*, 11(2):?–?, June 1986.

[724] Y. Sagiv. *Optimization of queries in relational databases*. PhD thesis, Princeton University, 1978.

[725] Y. Sagiv. *Optimization of Queries in Relational Databases*. UMI Research Press, Ann Arbor, Michigan, 1981.

[726] Y. Sagiv. Quadratic algorithms for minimizing joins in restricted relational expressions. *SIAM J. Comput.*, 12(2):321–346, 1983.

[727] Y. Sagiv and M. Yannakakis. Equivalence among expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, 1980.

[728] V. Sarathy, L. Saxton, and D. Van Gucht. Algebraic foundation and optimization for object based query languages. In *Proc. IEEE Conference on Data Engineering*, pages 113–133, 1993.

[729] C. Sartiani. A general framework for estimating xml query cardinality. In *Int. Workshop on Database Programming Languages*, pages 257–277, 2003.

[730] F. Scarcello, G. Greco, and N. Leone. Weighted hypertree decomposition and optimal query plans. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 210–221, 2004.

[731] J. Scheible. A survey of storage options. *IEEE Computer*, 35(12):42–46, 2002.

[732] H.-J. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[733] W. Scheufele. *Algebraic Query Optimization in Database Systems*. PhD thesis, Universität Mannheim, 1999.

[734] W. Scheufele and G. Moerkotte. Optimal ordering of selections and joins in acyclic queries with expensive predicates. Technical Report 96-3, RWTH-Aachen, 1996.

[735] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 238–248, 1997.

[736] W. Scheufele and G. Moerkotte. Efficient dynamic programming algorithms for ordering expensive joins and selections. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 201–215, 1998.

[737] J. Schindler, A. Ailamaki, and G. Ganger. Lachesis: Robust database storage management based on device-specific performance characteristics. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 706–717, 2003.

[738] J. Schindler and G. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.

[739] J. Schindler, J. Griffin, C. Lumb, and G. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. Technical Report CMU-CS-01-119, Carnegie Mellon University, 2001.

[740] J. Schindler, J. Griffin, C. Lumb, and G. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Conf. on File and Storage Technology (FAST)*, pages 259–274, 2002.

[741] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2000.

[742] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, 2001.

[743] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. on Database Systems*, 2(3):247–261, 1977.

[744] M. Scholl. Theoretical foundation of algebraic optimization utilizing unnormalized relations. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages ?–?, 1986.

[745] T. Schwentick. Xpath query containment. *ACM SIGMOD Record*, 33(1):101–109, 2004.

[746] E. Sciore and J. Sieg. A modular query optimizer generator. In *Proc. IEEE Conference on Data Engineering*, pages 146–153, 1990.

[747] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(4):387–407, 1996.

[748] B. Seeger, P.-A. Larson, and R. McFadyen. Reading a set of disk pages. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 592–603, 1993.

[749] A. Segev. Optimization of join operations in horizontally partitioned database systems. *ACM Trans. on Database Systems*, 11(1):48–80, 1986.

[750] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, 1979.

[751] T. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–185, 1988.

[752] T. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–186, 1988.

[753] T. Sellis. Multiple-query optimization. *ACM Trans. on Database Systems*, 13(1):23–52, 1988.

[754] T. K. Sellis. Global query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 191–205, 1986.

[755] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *USENIX*, pages 313–323, 1990.

[756] V. Sengar and J. Haritsa. PLASTIC: Reducing query optimization overheads through plan recycling. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, page 676, 2003.

[757] K. Seppi, J. Barnes, and C. Morris. A bayesian approach to database query optimization. *ORSA Journal of Computing*, 5(4):410–418, 1993.

[758] P. Seshadri, J. Hellerstein, H. Pirahesh, T. Leung, R. Ramakrishnan, D. Srivastava, P. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 435–446, 1996.

[759] P. Seshadri, H. Pirahesh, and C. Leung. Decorrelation of complex queries. Research Report RJ 9846 (86093), IBM Almaden Research Division, June 1994.

[760] P. Seshadri, H. Pirahesh, and T. Leung. Complex query decorrelation. In *Proc. IEEE Conference on Data Engineering*, pages 450–458, 1996.

[761] K. Sevcik. Data base system performance prediction using an analytical model. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 182–198, 1981.

[762] D. Severance. A practitioner's guide to data base compression. *Information Systems*, 8(1):51=62, 1983.

[763] D. Severance and G. Lohman. Differential files: their application to the maintenance of large databases. *ACM Trans. on Database Systems*, 1(3):256–267, Sep 1976.

[764] M. C. Shan. Optimal plan search in a rule-based query optimizer. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proc. of the Intl. Conf. on Extending Database Technology*, pages 92–112, Venice, Italy, Mar 1988. Springer-Verlag, Lecture Notes in Computer Science No. 303.

[765] J. Shanmugasundaram, R. Barr E. J. Shekita, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 65–76, 2000.

[766] G. Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 256–276, 1984.

[767] L. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-M. Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *IDEAS*, pages 20–33, 2001.

[768] L. Shapiro and A. Stephens. Bootstrap percolation, the schröder numbers and the n-kings problem. *SIAM J. Discr. Math.*, 4(2):275–280, 1991.

[769] A. Shatdal and J. Naughton. Processing aggregates in parallel database systems. Technical Report TR 1233, University of Wisconsin, 1994.

[770] G. M. Shaw and S.B. Zdonik. Object-oriented queries: Equivalence and optimization. In *1st Int. Conf. on Deductive and Object-Oriented Databases*, pages 264–278, 1989.

[771] G. M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. Tech. report no. cs-89-19, Department of Computer Science, Brown University, 1989.

[772] G.M. Shaw and S.B. Zdonik. An object-oriented query algebra. In *2nd Int. Workshop on Database Programming Languages*, pages 111–119, 1989.

[773] G.M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. In *Proc. IEEE Conference on Data Engineering*, pages 154–162, 1990.

[774] E. Shekita and M. Carey. A performance evaluation of pointer-based joins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 300–311, 1990.

[775] E. Shekita, H. Young, and K.-L. Tan. Multi-join optimization for symmetric multiprocessors. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 479–492, 1993.

[776] P. Shenoy and H. Cello. A disk scheduling framework for next generation operating systems. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 44–55, 1998.

[777] S. T. Shenoy and Z. M. Ozsoyoglu. A system for semantic query optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 181–195, 1987.

[778] S. Sherman and R. Brice. Performance of a database manager in a virtual memory system. *ACM Trans. on Database Systems*, 1(4):317–343, 1976.

[779] B. Shneiderman and V. Goodman. Batched searching of sequential and tree structured files. *ACM Trans. on Database Systems*, 1(3):208–222, 1976.

[780] E. Shriver. *Performance Modeling for Realistic Storage Devices*. PhD thesis, University of New York, 1997.

[781] E. Shriver, A. Merchant, and J. Wilkes. An analytical behavior model for disk drives with readahead caches and request reordering. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 182–191, 1998.

[782] A. Shrufi and T. Topaloglou. Query processing for knowledge bases using join indices. In *Int. Conference on Information and Knowledge Management (CIKM)*, 1995.

[783] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1996.

[784] K. Shwayder. Conversion of limited entry decision tables to computer programs — a proposed modification to Pollack's algorithm. *Communications of the ACM*, 14(2):69–73, 1971.

[785] M. Siegel, E. Sciore, and S. Salveter. A method for automatic rule derivation to support semantic query optimization. *ACM Trans. on Database Systems*, 17(4):53–600, 1992.

[786] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, 1997. 3rd Edition.

[787] D. Simmen, C. Leung, and H. Pirahesh. Exploitation of uniqueness properties for the optimization of SQL queries using a 1-tuple condition. Research Report RJ 10008 (89098), IBM Almaden Research Division, Feb. 1996.

[788] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 57–67, 1996.

[789] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 625–62, 1996.

[790] G. Slivinskas, C. Jensen, and R. Snodgrass. Bringing order to query optimization. *SIGMOD Record*, 13(2):5–14, 2002.

[791] D. Slutz. Massive stochastic testing of sql. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 618–622, 1998.

[792] D. Smith and M. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26:171–215, 1985.

[793] J. A. Smith. Sequentiality and prefetching in database systems. *ACM Trans. on Database Systems*, 3(3):223–247, 1978.

[794] J. M. Smith and P. Y.-T. Chang. Optimizing the performance of a relational algebra database interface. *Communications of the ACM*, 18(10):568–579, 1975.

[795] R. Sosic, J. Gu, and R. Johnson. The Unison algorithm: Fast evaluation of boolean expressions. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1:456 – 477, 1996.

[796] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. Patel, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. IEEE Conference on Data Engineering*, 2002.

[797] D. Srivastava, S. Dar, J. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 318–329, 1996.

[798] R. Stanley. *Enumerative Combinatorics, Volume I*, volume 49 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1997.

[799] H. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, 1995.

[800] H. Steenhagen, P. Apers, and H. Blanken. Optimization of nested queries in a complex object model. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 337–350, 1994.

[801] H. Steenhagen, P. Apers, H. Blanken, and R. de By. From nested-loop to join queries in oodb. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 618–629, 1994.

[802] H. Steenhagen, R. de By, and H. Blanken. Translating OSQL queries into efficient set expressions. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 183–197, 1996.

[803] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, Aug. 1997.

[804] M. Steinbrunn, K. Peithner, G. Moerkotte, and A. Kemper. Bypassing joins in disjunctive queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 228–238, 1995.

[805] M. Stillger and J.-C. Freytag. Testing the quality of a query optimizer. *IEEE Data Engineering Bulletin*, 18(3):41–48, Sept 1995.

[806] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO – DB2's learning optimizer. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 19–28, 2001.

[807] K. Stocker, D. Kossmann, R. Braumandl, and A. Kemper. Integrating semi-join reducers into state-of-the-art query processors. In *Proc. IEEE Conference on Data Engineering*, pages 575–584, 2001.

[808] L. Stockmeyer and C. Wong. On the number of comparisons to find the intersection of two relations. Technical report, IBM Watson Research Center, 1978.

[809] H. Stone and H. Fuller. On the near-optimality of the shortest-latency-time-first drum scheduling discipline. *Communications of the ACM*, 16(6):352–353, 1973.

[810] M. Stonebraker. Inclusion of new types in relational database systems. In *Proc. IEEE Conference on Data Engineering*, pages ?–?, 1986.

[811] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Trans. on Database Systems*, 12(3):350–376, Sep 1987.

[812] M. Stonebraker and P. Brown. *Object-Relational DBMSs, Tracking the Next Great Wave*. Morgan Kaufman, 1999.

[813] M. Stonebraker et al. QUEL as a data type. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Boston, MA, June 1984.

[814] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamios. On rules, procedures, caching and views in data base systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 281–290, 1990.

[815] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proc. of the 15nth ACM SIGMOD*, pages 340–355, 1986.

[816] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Trans. on Database Systems*, 1(3):189–222, 1976.

[817] D. Straube and T. Özsu. Access plan generation for an object algebra. Technical Report TR 90-20, Department of Computing Science, University of Alberta, June 1990.

[818] D. Straube and T. Özsu. Queries and query processing in object-oriented database systems. Technical report, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1990.

[819] D. Straube and T. Özsu. Queries and query processing in object-oriented database systems. *ACM Trans. on Information Systems*, 8(4):387–430, 1990.

[820] D. Straube and T. Özsu. Execution plan generation for an object-oriented data model. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 43–67, 1991.

[821] D. D. Straube. *Queries and Query Processing in Object-Oriented Database Systems*. PhD thesis, The University of Alberta, Edmonton, Alberta, Canada, Dec 1990.

[822] S. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient views. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 319–330, Seattle, WA, 1998.

[823] D. Suciu. Query decomposition and view maintenance for query languages for unconstrained data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 227–238, 1996.

[824] N. Südkamp and V. Linnemann. Elimination of views and redundant variables in an SQL-like database language for extended NF^2 structures. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 302–313, 1990.

[825] K. Sutner, A. Satyanarayana, and C. Suffel. The complexity of the residual node connectedness reliability problem. *SIAM J. Comp.*, 20(1):149–155, 1991.

[826] P. Svensson. On search performance for conjunctive queries in compressed, fully transposed ordered files. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 155–163, 1979.

[827] A. Swami. *Optimization of Large Join Queries*. PhD thesis, Stanford University, 1989. Technical Report STAN-CS-89-1262.

[828] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 367–376, 1989.

[829] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 8–17, 1988.

[830] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. Technical Report RJ 8812, IBM Almaden Research Center, 1992.

[831] A. Swami and B. Iyer. A polynomial time algorithm for optimizing join queries. In *Proc. IEEE Conference on Data Engineering*, pages 345–354, 1993.

[832] A. Swami and B. Schiefer. Estimating page fetches for index scans with finite LRU buffers. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 173–184, 1994.

[833] A. Swami and B. Schiefer. On the estimation of join result sizes. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 287–300, 1994.

[834] N. Talagala, R. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based extraction of local and global disk characteristics. Technical Report UCB-CSD-99-1063, University of Berkeley, 2000.

[835] K.-L. Tan and H. Lu. A note on the strategy space of multiway join query optimization problem in parallel systems. *SIGMOD Record*, 20(4):81–82, 1991.

[836] A. Tansel and L. Garnett. On roth, korth, and silberschatz's extended algebra and calculus for nested relational databases. *ACM Trans. on Database Systems*, 17(2):374–383, 1992.

[837] Y. C. Tay. On the optimality of strategies for multiple joins. *Journal of the ACM*, 40(5):1067–1086, 1993.

[838] T. Teorey and T. Pinkerton. A comparative analysis of disk scheduling policies. In *Proc. of the AFIPS Fall Joint Computer Conference*, pages 1–11, 1972.

[839] T. Teorey and T. Pinkerton. A comparative analysis of disk scheduling policies. *Communications of the ACM*, 15(3):177–184, 1972.

[840] J. Teubner, T. Grust, and M. Van Keulen. Bridging the gap between relational and native xml storage with staircase join. *Grundlagen von Datenbanken*, pages 85–89, 2003.

[841] C. Tompkins. Machine attacks o problems whose variables are permutations. *Numerical Analysis (Proc. of Symposia in Applied Mathematics)*, 6, 1956.

[842] R. Topor. Join-ordering is NP-complete. Draft, personal communication, 1998.

[843] Transaction Processing Council (TPC). TPC Benchmark D. http://www.tpc.org, 1995.

[844] Transaction Processing Performance Council, 777 N. First Street, Suite 600, San Jose, CA, USA. *TPC Benchmark R*, 1999. Revision 1.2.0. http://www.tpc.org.

[845] P. Triantafillou, S. Christodoulakis, and C. Georgiadis. A comprehensive analytical performance model for disk devices under random workloads. *IEEE Trans. on Knowledge and Data Eng.*, 14(1):140–155, 2002.

[846] O. Tsatalos, M. Solomon, and Y. Ioannidis. The GMAP: A versatile tool for physical data independence. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 367–378, 1994.

[847] K. Tufte and D. Maier. Aggregation and accumulation of XML data. *IEEE Data Engineering Bulletin*, 24(2):34–39, 2001.

[848] J.D. Ullman. *Database and Knowledge Base Systems*, volume Volume 1. Computer Science Press, 1989.

[849] J.D. Ullman. *Database and Knowledge Base Systems*, volume Volume 2. Computer Science Press, 1989.

[850] J.D. Ullman. *Database and Knowledge Base Systems*. Computer Science Press, 1989.

[851] D. Straube und T. Özsu. Query transformation rules for an object algebra. Technical Report TR 89-23, Department of Computing Science, University of Alberta, Sept. 1989.

[852] T. Urhan, M. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 130–141, 1998.

[853] M. Uysal, G. Alvarez, and A. Merchant. A modular analytical throughput model for modern disk arrays. In *MASCOTS*, pages 183–192, 2001.

[854] P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, 1987.

[855] P. Valduriez and H. Boral. Evaluation of recursive queries using join indices. In *Proc. Int. Conf. on Expert Database Systems (EDS)*, pages 197–208, 1986.

[856] P. Valduriez and S. Danforth. Query optimization in database programming languages. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 516–534, 1989.

[857] L. Valiant. The complexity of computing the permanent. *Theoretical Comp. Science*, 8:189–201, 1979.

[858] L. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comp.*, 8(3):410–421, 1979.

[859] B. Vance. *Join-order Optimization with Cartesian Products*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.

[860] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, 1996.

[861] S. L. Vandenberg and D. DeWitt. An algebra for complex objects with arrays and identity. Internal report, Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA, 1990.

[862] S. L. Vandenberg and D. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 158–167, 1991.

[863] J. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 193–204, 1999.

[864] G. von Bültzingsloewen. Translating and optimizing sql queries having aggregates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 235–243, 1987.

[865] G. von Bültzingsloewen. *Optimierung von SQL-Anfragen für parallele Bearbeitung (Optimization of SQL-queries for parallel processing)*. PhD thesis, University of Karlsruhe, 1990. in German.

[866] G. von Bültzingsloewen. *SQL-Anfragen: Optimierung für parallele Bearbeitung*. FZI-Berichte Informatik. Springer, 1991.

[867] F. Waas and C. Galindo-Legaria. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. Technical Report INS-R-9913, CWI, Amsterdam, 1999.

[868] F. Waas and C. Galindo-Legaria. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 499–509, 2000.

[869] F. Waas and A. Pellenkoft. Probabilistic bottom-up join order selection – breaking the curse of NP-completeness. Technical Report INS-R9906, CWI, 1999.

[870] F. Waas and A. Pellenkoft. Join order selection - good enough is easy. In *BNCOD*, pages 51–67, 2000.

[871] J. Wang, J. Li, and G. Butler. Implementing the PostgreSQL query optimzier within the OPT++ framework. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 262–272, 2003.

[872] J. Wang, M. Maher, and R. Topor. Rewriting unions of general conjunctive queries using views. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 52–69, 2002.

[873] M. Wang, J. Vitter, and B. Iyer. Selectivity estimation in the presence of alphanumeric correlations. In *Proc. IEEE Conference on Data Engineering*, pages 169–180, 1997.

[874] W. Wang, H. Jiang, H. Lu, and J. Yu. Containment join size estimation: Models and methods. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 145–156, 2003.

[875] W. Wang, H. Jiang, H. Lu, and J. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 240–251, 2004.

[876] X. Wang and M. Cherniack. Avoiding ordering and grouping in query processing. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 826–837, 2003.

[877] S. Waters. File design fallacies. *The Computer Journal*, 15(1):1–4, 1972.

[878] S. Waters. Hit ratio. *Computer Journal*, 19(1):21–24, 1976.

[879] H. Wedekind and G. Zörntlein. Prefetching in realtime database applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 215–226, 1986.

[880] M. Wedekind. On the selection of access paths in a database system. In J. Klimbie and K. Koffeman, editors, *IFIP Working Conference Data Base Management*, pages 385–397, Amsterdam, 1974. North-Holland.

[881] G. Weikum. Set-oriented disk access to large complex objects. In *Proc. IEEE Conference on Data Engineering*, pages 426–433, 1989.

[882] G. Weikum, B. Neumann, and H.-B. Paul. Konzeption und Realisierung einer mengenorientierten Seitenschnittstelle zum effizienten Zugriff auf komplexe Objekte. In *Proc. der GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft (BTW)*, pages 212–230, 1987.

[883] T. Westmann. *Effiziente Laufzeitsysteme für Datenlager*. PhD thesis, University of Mannheim, 2000.

[884] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. Technical Report 03/98, University of Mannheim, 1998.

[885] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[886] T. Westmann and G. Moerkotte. Variations on grouping and aggregations. Technical Report 11/99, University of Mannheim, 1999.

[887] K.-Y. Whang and R. Krishnamurthy. Query optimization in a memory-resident domain relational calculus database system. *ACM Trans. on Database Systems*, 15(1):67–95, Mar 1990.

[888] K.-Y. Whang, A. Malhotra, G. Sockut, and L. Burns. Supporting universal quantification in a two-dimensional database query language. In *Proc. IEEE Conference on Data Engineering*, pages 68–75, 1990.

[889] K.-Y. Whang, G. Wiederhold, and D. Sagalowicz. Estimating block accesses in database organizations: A closed noniterative formula. *Communications of the ACM*, 26(11):940–944, 1983.

[890] N. Wilhelm. A general model for the performance of disk systems. *Journal of the ACM*, 24(1):14–31, 1977.

[891] D. E. Willard. Efficient processing of relational calculus queries using range query theory. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 164–175, 1984.

[892] C. Williams and T. Hogg. Using deep structure to locate hard problems. In *Proc. National Conference on Artificial Intelligence*, pages 472–477, 1992.

[893] J. Wolf, R. Iyer, K. Pattipati, and J. Turek. Optimal buffer partitioning for the nested block join algorithm. In *Proc. IEEE Conference on Data Engineering*, pages 510–519, 1991.

[894] C. Wong. Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems. *ACM Computing Surveys*, 12(2):167–177, 1980.

[895] F. Wong and K. Youssefi. Decomposition – a strategy for query processing. *ACM Trans. on Database Systems*, 1(3):223–241, 1976.

[896] H. Wong and J. Li. Transposition algorithms on very large compressed databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 304–311, 1986.

[897] P. Wood. On the equivalence of XML patterns. In *CL 2000*, 2000.

[898] P. Wood. Minimizing simple XPath expressions. In *Int. Workshop on Database Programming Languages*, pages 13–18, 2001.

[899] P. Wood. Containment for xpath fragments under dtd constraints. In *Proc. Int. Conf. on Database Theory (ICDT)*, pages 300–314, 2003.

[900] W. A. Woods. Procedural semantics for question-answering systems. In *FJCC (AFIPS Vol. 33 Part I)*, pages 457–471, 1968.

[901] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. Scheduling algorithms for modern disk drives. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 241–251, 1994.

[902] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 146–156, 1995.

[903] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. Technical Report CSE-TR-323-96, University of Michigan, 1996.

[904] M.-C. Wu. Query optimization for selections using bitmaps. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 227–238, 1999.

[905] Y. Wu, J. Patel, and H.V. Jagadish. Estimating answer sizes for XML queries. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 590–608, 2002.

[906] Y. Wu, J. Patel, and H.V. Jagadish. Estimating answer sizes for XML queries. *Information Systems*, 28(1-2):33–59, 2003.

[907] Z. Xie. Optimization of object queries containing encapsulated methods. In *Proc. 2nd. Int. Conf. on Information and Knowledge Management*, pages 451–460, 1993.

[908] G. D. Xu. Search control in semantic query optimization. Technical Report 83-09, COINS, University of Massachusetts, Amherst, MA, 1983.

[909] W. Yan. *Rewriting Optimization of SQL Queries Containing GROUP-BY*. PhD thesis, University of Waterloo, 1995.

[910] W. Yan and P.-A. Larson. Performing group-by before join. Technical Report CS 93-46, Dept. of Computer Science, University of Waterloo, Canada, 1993.

[911] W. Yan and P.-A. Larson. Performing group-by before join. In *Proc. IEEE Conference on Data Engineering*, pages 89–100, 1994.

[912] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 345–357, 1995.

[913] W. Yan and P.-A. Larson. Interchanging the order of grouping and join. Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada, 1995.

[914] H. Yang and P.-A. Larson. Query transformation for PSJ-queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 245–254, 1987.

[915] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 136–145, 1997.

[916] Qi Yang. Computation of chain queries in distributed database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 348–355, 1994.

[917] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 82–94, 1981.

[918] B. Yao and T. Özsu. XBench – A Family of Benchmarks for XML DBMSs. Technical Report CS-2002-39, University of Waterloo, 2002.

[919] B. Yao, T. Özsu, and N. Khandelwal. Xbench benchmark and performance testing of XML DBMSs. In *Proc. IEEE Conference on Data Engineering*, pages 621–632, 2004.

[920] S. B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, 1977.

[921] S. B. Yao. An attribute based model for database access cost analysis. *ACM Trans. on Database Systems*, 2(1):45–67, 1977.

[922] S. B. Yao and D. DeJong. Evaluation of database access paths. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 66–77, 1978.

[923] S.B. Yao. Optimization of query evaluation algorithms. *ACM Trans. on Database Systems*, 4(2):133–155, 1979.

[924] S.B. Yao, A.R. Hevner, and H. Young-Myers. Analysis of database system architectures using benchmarks. *IEEE Trans. on Software Eng.*, SE-13(6):709–725, 1987.

[925] Y. Yoo and S. Lafortune. An intelligent search method for query optimization by semijoins. *IEEE Trans. on Knowledge and Data Eng.*, 1(2):226–237, June 1989.

[926] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. Xrel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, June 2001.

[927] K. Youssefi and E. Wong. Query processing in a relational database management system. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 409–417, 1979.

[928] C. T. Yu, W. S. Luk, and M. K. Siu. On the estimation of the number of desired records with respect to a given query. *ACM Trans. on Database Systems*, 3(1):41–56, 1978.

[929] L. Yu and S. L. Osborn. An evaluation framework for algebraic object-oriented query models. In *Proc. IEEE Conference on Data Engineering*, 1991.

[930] J. Zahorjan, B. Bell, and K. Sevcik. Estimating block transfers when record access probabilities are non-uniform. *Information Processing Letters*, 16(5):249–252, 1983.

[931] B. T. Vander Zander, H. M. Taylor, and D. Bitton. Estimating block accesses when attributes are correlated. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 119–127, 1986.

[932] B. T. Vander Zander, H. M. Taylor, and D. Bitton. A general framework for computing block accesses. *Information Systems*, 12(2):177–?, 1987.

[933] S. Zdonik and G. Mitchell. ENCORE: An object-oriented approach to database modelling and querying. *IEEE Data Engineering Bulletin*, 14(2):53–57, June 1991.

[934] N. Zhang, V. Kacholia, and T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. IEEE Conference on Data Engineering*, pages 54–65, 2004.

[935] N. Zhang and T. Özsu. Optimizing correlated path expressions in XML languages. Technical Report CS-2002-36, University of Waterloo, 2002.

[936] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 159–170, 1997.

[937] Y. Zhao, P. Deshpande, J. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–282, 1998.

[938] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehouse environment. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1995.

Appendix F

ToDo

- size of a query in rel alg: [698]
- [872]
- Integrating Buffer Issues into Query Optimization: [207, 462]
- Integrating concurrency control issues into query optimization: [602, 603]
- [92]
- where do we put ”counting page accesses”?
- control, A^*, ballooning: [594, 593]
- Bypass Plans
- Properties (rather complete list, partial ordering, plan independent properties: store them somewhere else (dpstructure or memostructure))
- describe prep-phase of plan generator
- reuse plans: [756]
- estimating query compilation time: [431]
- cost model [761]
- sensivitity of QO to storage access cost parameters [696] (and join selectivities on join order: [505] [papier ist nicht ernst zu nehmen])
- magic set and semi join reducers [74, 76, 75, 165, 332, 611, 609, 611, 610, 758, 807, 925]
- join indexes and clustering tuples of different relations with 1:n relationship [224, 395, 854, 855, 782]
- B-Trees with space filling curves (Bayer)
- Prefetching [879]

- feedback to optimizer [495]
- compression [29, 51, 155, 204, 249, 248, 327, 333] [621, 691, 713, 762, 763, 826, 884, 896]
- semantic QO SQO: [1, 78, 161, 323, 355, 484, 485, 501, 508] [622, 630, 631, 653, 777, 785, 787, 908] [537]
- join processing with nonclustered indexes: [623]
- join+buffer: [893]
- removal/elimination of redundant joins [632, 824]
- benchmark(ing): Gray Book: [356]; papers: [89, 97, 640, 719, 742, 924, 918, 919]
- dynamic qo: [644] [31, 852] [41] [463]
- unnesting: [648, 689]
- prefetching: [652, 651, 793, 879]
- Starburst: [661, 662]
- BXP: [96, 250, 294, 366, 391, 427, 479, 666, 695, 784, 792, 795, 478]
  BXP complexity: [72] BXP var infl: [464]
- joins: [672]
- query folding: [674]
- quantification: [103, 102, 182, 183, 687, 679, 888] [455]
- outerjoins: [80, 81, 213, 292, 284, 283, 688, 704]
- partial match + hashing: [682]
- OODB indexing by class division: [189, 686]
- decision support [690]
- tree structured databases: [697]
- Rosenthal: [707, 692, 708, 709, 693, 703, 706, 705]
- conj. queries [701]
- aggregation/(generalized proj): [106, 189, 285, 712] [371, 372, 400]
- do nest/unnest to optimize duplicate work: [710]
  $e_1 \bowtie_{A_1=A_2} e_2 \equiv \mu_g(e_1 \bowtie_{A_1=A_2} \Gamma_{g;=A_2;id}(e_2))$
- join size: [702]
- fragmentation: [723]

- eqv: [23, 22]
- alg eqvs union/difference: [727] [872]
- other sagiv: [725, 726]
- bayesian approach to QO: [757]
- cache query plans: [756]
- joins for horizontally fragmentation: [749]
- partitioning: [52, 95, 389, 467, 625]
- MQO: [27, 124, 122, 754, 753, 937]
- indexing+caching: [752]
- rule-based QO: [764, 62, 63, 280]
- rule-based IRIS: [223]
- cost: [808] [853]
- search space: [835], join ordering: [837]
- access path: [118, 880, 922, 90]
- eff aggr: [282] [886]
- misc: [891] [11] [15]
- access paths: bitmaps [904]
- dist db: [36, 37, 88, 211, 916] Donald's state of the art: [498]
- [132, 133]
- eqv: bags [25, 216]
- eqvs old: [26]
- DB2: norwegian analysis: [32]
- nested: [44]
- Genesis/Praire/Batory: [53, 57, 56, 58, 212]
- eqvs OO: [64, 65]
- dupelim: [83]
- (generalized) division: [116, 210, 345, 335]
- early aggregation
- chunks-wise processing [225, 342]

- temporal intersection join: [368]
- 2nd ord sig: Güting: [376]
- classics: [386]
- smallest first: [390]
- Hwang/Yu: [426]
- Kambayashi: [466]
- Koch [493], Lehnert [527]
- I/O cost reduction for (hash) joins: [551, 587]
- dist nest: [255]
- band join: [556]
- Donovan (TODS 76,1,4) Decision Support: [239]
- whenever materialize something (sort, hash join, etc) compute min/max of some attributes and use these as additional selection predicates
- determine optimal page access sequence and buffer size to access pairs (x,y) of pages where join partners of one relation lie on x and of the other on y (Fotouhi, Pramanik [277], Merret, Kambayashi, Yasuura [587], Omiecinski [623], Pramanik, Ittner [672], Chan, Ooi [127])
- Scarcello, Greco, Leone: [730]
- Sigmod05:
  - proactive reoptimization [43]
  - robust query optimizer [42]
  - stacked indexed views [220]
  - NF^2-approach to processing nested sql queries [109]
  - efficient computatio of multiple groupby queries [158]
- LOCI: [643]
- Wavelet synopses: [312]
- Progress Indicators: [140, 558, 559]
- PostgresExperience: [871]
- Chaudhuri sigmod 05: [42]