Execution Strategies for SQL Subqueries

Mostafa Elhemali, **César Galindo-Legaria**, Torsten Grabs, Milind Joshi

Microsoft Corp

With additional slides from material in paper, added by S. Sudarshan

1

Motivation

- Optimization of subqueries has been studied for some time
- Challenges
  - Mixing scalar and relational expressions
  - Appropriate abstractions for correct and efficient processing
  - Integration of special techniques in complete system
- This talk presents the approach followed in SQL Server
- Framework where specific optimizations can be plugged
- Framework applies also to “nested loops” languages

2

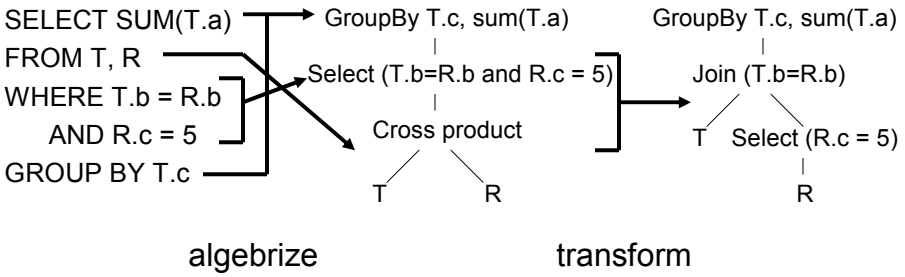
Outline

- **Query optimizer context**
- Subquery processing framework
- Subquery disjunctions

3

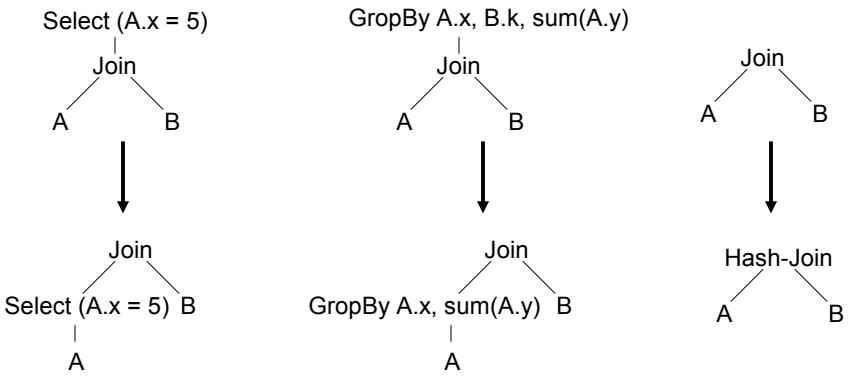
Algebraic query representation

- Relational operator trees
- **Not** SQL-block-focused



4

Operator tree transformations



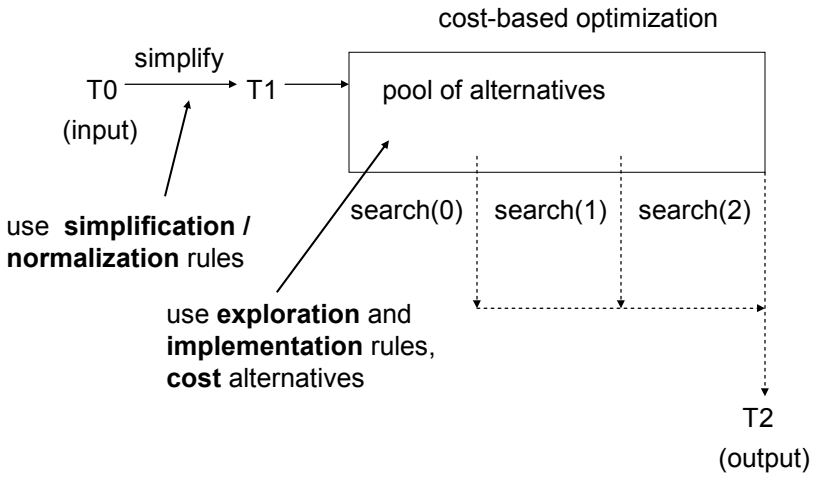
Simplification / normalization

Exploration

Implementation

5

SQL Server Optimization process



6

Plan Generation Overview

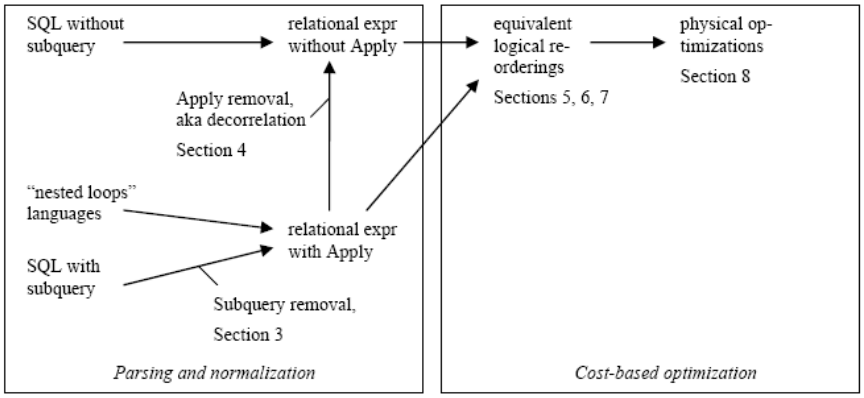


Figure 1: Overview of plan generation for subqueries in SQL Server and structure of the paper

Outline

- Query optimizer context
- **Subquery processing framework**
- Subquery disjunctions

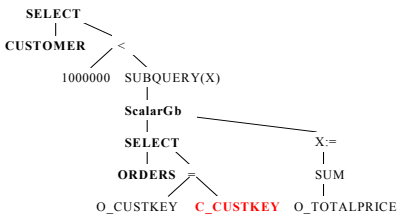
SQL Subquery

- A relational expression where you expect a scalar
  - Existential test, e.g. NOT EXISTS(SELECT…)
  - Quantified comparison, e.g. T.a =ANY (SELECT…)
  - Scalar-valued, e.g. T.a = (SELECT…) + (SELECT…)
- Convenient and widely used by query generators

9

Algebrization

```
select *
from customer
where 100,000 <
    (select sum(o_totalprice)
     from orders
     where o_custkey = c_custkey)
```



Subqueries: relational operators with scalar parents
Commonly "correlated," i.e. they have outer-references

10

Subquery removal

- Executing subquery requires mutual recursion between scalar engine and relational engine
- **Subquery removal**: Transform tree to remove relational operators from under scalar operators
- Preserve special semantics of using a relational expression in a scalar, e.g. at-most-one-row

11

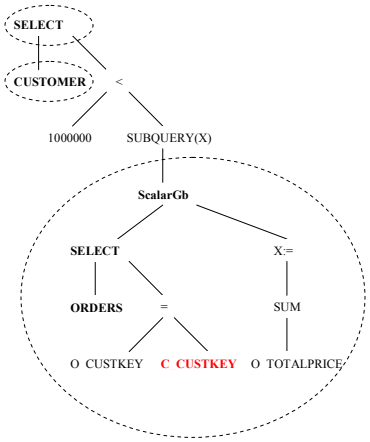
The Apply operator

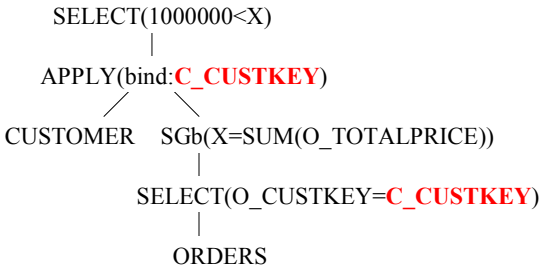
$$R\ \mathcal{A}pply_{\mathcal{JN}}\ E(\boldsymbol{r}) = \mathcal{UA}_{r \in R}\ (\{r\}\ \mathcal{JN}\ E(\boldsymbol{r}))$$

- R **Apply** E(**r**)
  - For each row r of R, execute function E on r
  - Return union: $\{r_1\}$ X $E(r_1)$ U $\{r_2\}$ X $E(r_2)$ U …
  - Abstracts "for each" and relational function invocation
- Also known as d-join and tuple-substitution join
- Variants: left outer join, semi-join, anti-join
- Exposed in SQL Server (FROM clause)
  - LATERAL clause in SQL standard
- Useful to invoke table-valued functions

12

Subquery removal





13

Algebraization of SubQueries

- SQL Query:
  - SELECT *, (SELECT C_NAME FROM CUSTOMER WHERE C_CUSTKEY = **O_CUSTKEY**) FROM ORDERS
- Translated to
  - ORDERS $Apply_{OJ}$ (π [C_NAME] σ [C_CUSTKEY = **O_CUSTKEY**] CUSTOMER)
- In general:
  - R $Apply_{OJ}$ *max1row(E(**r**))*
- Subqueries with exists/not exists become
  - R $Apply_{SJ}$ *E(**r**)*
  - R $Apply_{ASJ}$ *E(**r**)*

14

Conditional Scalar Execution

- Expression
  - CASE WHEN EXISTS(*E1(**r**)*) THEN *E2(**r**)* ELSE 0 END
- Translated to
  - $\pi_{\text{CASE WHEN } p = 1 \text{ THEN } e2 \text{ ELSE 0 END}}$ (
    (R *Apply[semijoin, probe as* p*] E1(**r**))*
    Apply[outerjoin, pass-through p*=1]*
    *max1row(E2(**r**)) as e2)*

15

Disjunction of SubQueries

- *WHERE p(**r**)* OR EXISTS(*E1(**r**)*) OR EXISTS(*E2(**r**)*)
- R $Apply_{SJ}$ *((*$\sigma_{p(r)}$ CT(1) *UA E1(**r**) UA E2(**r**))*
  - CT(1): Constant Table returning 1
  - UA: Union All
- Can also translate to apply with passthrough

16

Quantification and NULLs

- Consider predicate 5 NOT IN S which is equivalent to <>ALL
- The result of this predicate is as follows, for various cases of set S:
  1. If S = {} then p is TRUE.
  2. If S = {1} then p is TRUE.
  3. If S = {5} then p is FALSE.
  4. If S = {NULL, 5} then p is FALSE.
  5. If S = {NULL, 1} then p is UNKNOWN.
- (FOR ALL s in S: p)
  - = (NOT EXISTS s in S: NOT p): But only without nulls
  - In general predicate A cmp B is translated as
    A <cmp'> B OR A IS NULL OR B IS NULL
    - where cmp' is the complement of cmp

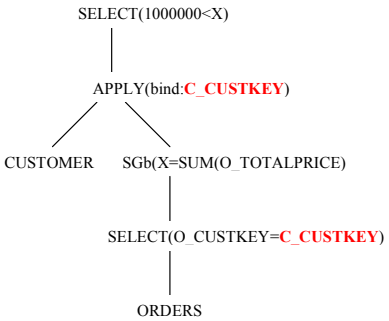
17

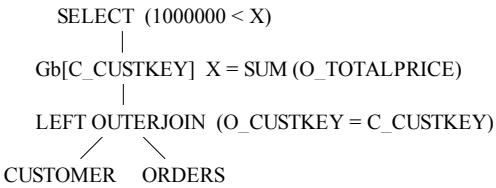
Apply removal

- Executing Apply forces nested loops execution into the subquery
- **Apply removal**: Transform tree to remove Apply operator
- The crux of efficient processing
- Not specific to SQL subqueries
- Can go by "unnesting," "decorrelation," "unrolling loops"
- Get joins, outerjoin, semijoins, … as a result

18

Apply removal





Apply does not add expressive power to relational algebra
Removal rules exist for different operators

19

Why remove Apply?

- Goal is **NOT** to avoid nested loops execution, but to normalize the query
- Queries formulated using “for each” surface may be executed more efficiently using set-oriented algorithms
- … and queries formulated using declarative join syntax may be executed more efficiently using nested loop, “for each” algorithms

20

Removing Apply Cont.

- Apply removal that preserves the size of the expression.
  - With Apply
    - ORDERS $Apply_{OJ}$ (σ[C_CUSTKEY = **O_CUSTKEY**] CUSTOMER)
  - Removing apply
    - ORDERS *OJ* [C_CUSTKEY = O_CUSTKEY] CUSTOMER
- Apply removal that duplicates subexpressions.

$$R\ \mathcal{A}pply_{\mathcal{JN}}\ (\sigma_{\boldsymbol{R.a}\,=\,S.a}\ S)\ \mathcal{UA}\ \sigma_{\boldsymbol{R.b}\,=\,T.b}\ T)$$

$$= R\ \mathcal{JN}_{R.a\,=\,S.a}\ S\ \mathcal{UA}\ R\ \mathcal{JN}_{R.b\,=\,T.b}\ T$$

- Apply removal not always possible
  - max1row/pass-through predicates, opaque functions

21

Magic Sets

- Originally formulated for recursive query processing
- Special case for non-recursive queries

$$R\ \mathcal{A}pply\ E(\boldsymbol{r}) = R\ \mathcal{JN}_{R.r=R'.r'}\ DS$$

$$DS = R'\ \mathcal{A}pply\ E(r')$$

$$R' = distinct[R.r]\ R$$

22

Magic Sets with Group By

A: $R \mathcal{JN}_{p(r,s) \text{ and } p(r,x)} \mathcal{G}_{s,x=agg} S$

M: $R \mathcal{JN}_{r = r' \text{ and } p(r,x)} (\mathcal{G}_{s,x=agg} R' \mathcal{JN}_{p(r',s)} S)$

- Other options
  - B: Pull groupby above join
  - C: "**Segmented execution**", when R and S are the same
    - E.g. Select all students with the highest mark

23

Reordering Semijoins and Antijoins

- Pushing down semi/anti joins

$$(\mathcal{G}_{A,F} R) \mathcal{SJ}_{p(A,S)} S = \mathcal{G}_{A,F} (R \mathcal{SJ}_{p(A,S)} S)$$

$$(\mathcal{G}_{A,F} R) \mathcal{ASJ}_{p(A,S)} S = \mathcal{G}_{A,F} (R \mathcal{ASJ}_{p(A,S)} S)$$

- Converting semi-join to join (to allow reordering)

$$R \mathcal{SJ}_{p(R,S)} S = \mathcal{G}_{key(R),Any(R)} (R \mathcal{Join}_{p(R,S)} S)$$

$$\text{ORDERS } \mathcal{SJ} (\text{CUSTOMER } \mathcal{JN} \text{ SUPPLIER } \mathcal{JN} \text{ LINEITEM})$$

  - How about anti-joins?

24

Subquery Disjunctions

```
SELECT COUNT(*)
FROM LINEITEM
WHERE L_SHIPDATE != ALL(SELECT O_ORDERDATE
from ORDERS)
```

- generates an antijoin with predicate

  L_SHIPDATE = O_ORDERDATE OR L_SHIPDATE IS NULL

- which can be rewritten using

  $$(R\ \mathcal{ASJ}_{(p1\ \mathrm{OR}\ p2)}\ S) = ((R\ \mathcal{ASJ}_{p1}\ S)\ \mathcal{ASJ}_{p2}\ S)$$

- Another useful rule

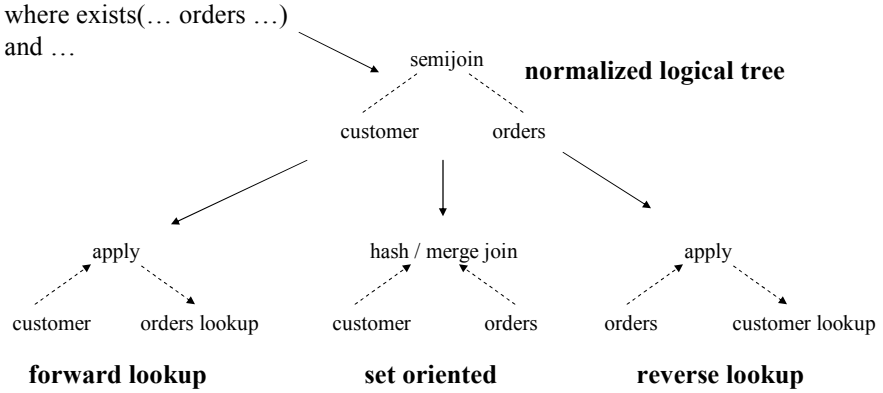
  $$R\ \mathcal{Apply}_{SJ}\,(E_1(\mathbf{r_1})\ \mathcal{UA}\ E_2(\mathbf{r_2})\ \mathcal{UA}\ E_3(\mathbf{r_3})\ \ldots) \rightarrow$$

  $$(R\ \mathcal{Apply}_{SJ}\ E_1(\mathbf{r_1}))\ \mathcal{U}\ (R\ \mathcal{Apply}_{SJ}\ E_2(\mathbf{r_2}))\ \mathcal{U}\ (R\ \mathcal{Apply}_{SJ}\ E_3(\mathbf{r_3}))\ \ldots$$

25

Categories of execution strategies

select …
from customer
where exists(… orders …)
and …



26

Forward lookup

APPLY[semijoin](bind:**C_CUSTKEY**)

CUSTOMER ORDERS Lkup(O_CUSTKEY=**C_CUSTKEY**)

The "natural" form of subquery execution
Early termination due to semijoin – pull execution model
Best alternative if few CUSTOMERs and index on ORDER exists

27

Reverse lookup

DISTINCT on C_CUSTKEY

APPLY(bind:**O_CUSTKEY**)

ORDERS CUSTOMERS Lkup(C_CUSTKEY=**O_CUSTKEY**)

APPLY(bind:**O_CUSTKEY**)

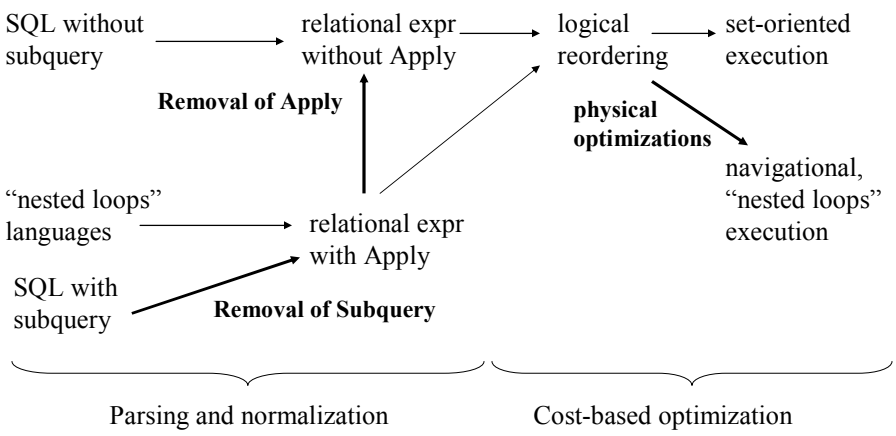
DISTINCT on O_CUSTKEY CUSTOMERS Lkup(C_CUSTKEY=**O_CUSTKEY**)

ORDERS

Mind the duplicates
Consider reordering GroupBy (DISTINCT) around join

28

Subquery processing overview



29

The fine print

- Can you always remove subqueries?
  - Yes, but you need a quirky Conditional Apply
  - Subqueries in CASE WHEN expressions
- Can you always remove Apply?
  - Not Conditional Apply
  - Not with opaque table-valued functions
- Beyond yes/no answer: Apply removal can explode size of original relational expression

30

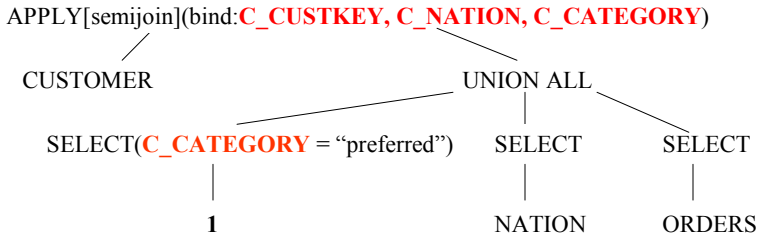
Outline

- Query optimizer context
- Subquery processing framework
- **Subquery disjunctions**

31

Subquery disjunctions

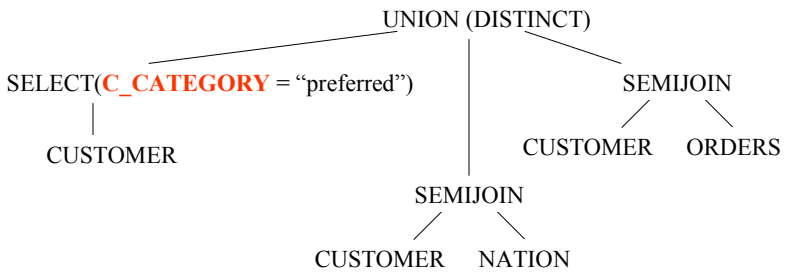
select * from customer
where c_catgory = “preferred”
or exists(select * from nation where n_nation = **c_nation** and …)
or exists(select * from orders where o_custkey = **c_custkey** and …)



Natural forward lookup plan
Union All with early termination short-circuits “OR” computation

32

Apply removal on Union



Distributivity replicates outer expression
Allows set-oriented and reverse lookup plan
This form of Apply removal done in cost-based optimization, not simplification

33

Optimizing Apply

- Caching of results from earlier calls
  - Trivial if no correlation variables
  - In-memory if few distinct values/small results
  - May or may no be worthwhile if large results
- Asynchronous IO
- Batch Sort

34

Asynchronous IO

- Ask OS to prefetch data, continue doing other things while prefetch is happening
  - better use of resources, esp with multiple disks/CPUs
- SELECT <blah blah>
  FROM PART natural join SUPPLIER natural join PARTSUPP
  WHERE <restrictive selections>
  AND PS_SUPPLYCOST = (SELECT MIN(PS_SUPPLYCOST)
  FROM PARTSUPP, SUPPLIER
  WHERE **P_PARTKEY** = PS_PARTKEY
  AND S_SUPPKEY = PS_SUPPKEY)
- Plan used by SQL Server (how the hell did it come up with this?)

$$R\ \mathcal{A}pply_{JN}\ (\sigma[\mathbf{R.m} = \text{PS_SUPPLYCOST}]\ (Seek_{R.P_PARTKEY=PS_PARTKEY}\ \text{PARTSUPP}))$$

  - where

$$R = \mathcal{G}_{P_PARTKEY;m=min(PS_SUPPLYCOST)}\ (\text{PART}\ \mathcal{JN}\ \text{SUPPLIER}\ \mathcal{JN}\ \text{PARTSUPP})$$

35

Batch Sort

- Sort order of parameters can help inner query
- But sorting outer query can be time consuming
  - esp if we stop after a few answers
- So: batch a group of outer parameters, sort them, then invoke inner in sorted order
- Batch size increased step-wise
  - so first few answers are fast at cost of more IO, later ones optimize IO more but with some delay

36

Summary

- Presentation focused on overall framework for processing SQL subqueries and "for each" constructs
- Many optimization pockets within such framework – you can read in the paper:
  - Optimizations for semijoin, antijoin, outerjoin
  - "Magic subquery decorrelation" technique
  - Optimizations for general Apply
  - …
- Goal of "decorrelation" is not set-oriented execution, but to normalize and open up execution alternatives

37

A question of costing

Fwd-lookup … 10ms to 3 days

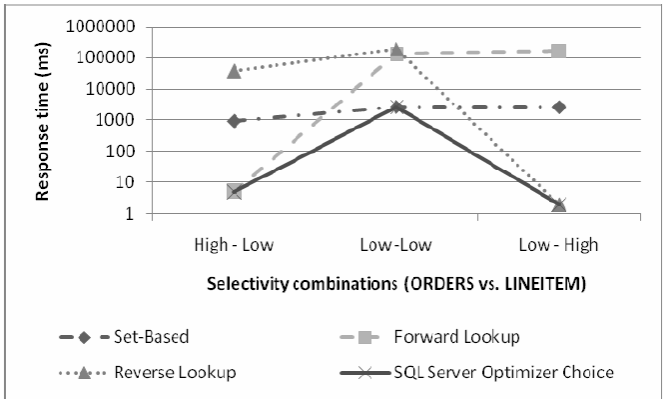
Bwd-lookup … 10ms to 3 days, cases opposite to fwd-lkp
Optimizer that picks the right strategy for you … priceless

Set-oriented execution … 2 to 3 hours

38

Execution Strategies for Semijoin

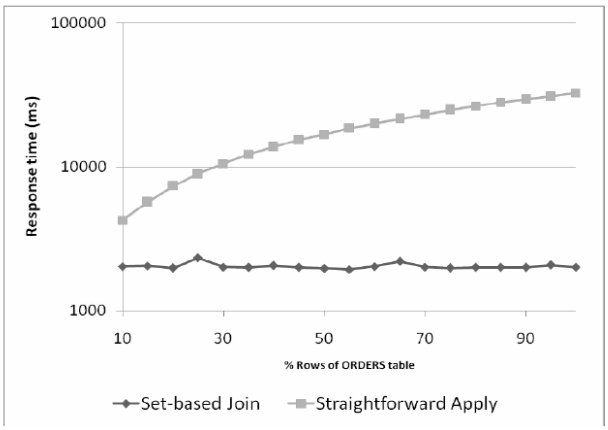
- Outer query on orders, exists subquery on lineitem (Section 6.1)



39

Execution Strategies for Antijoin

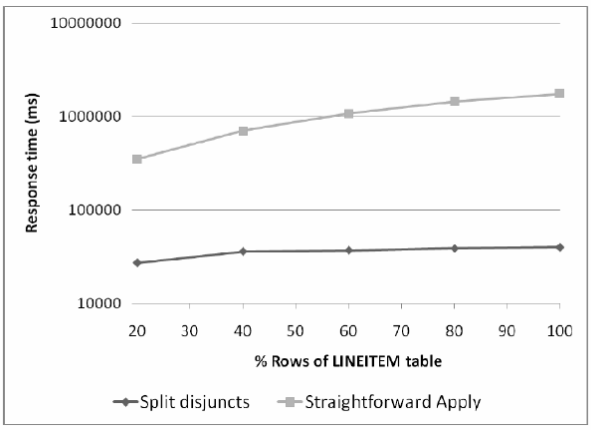
- Outer query uses only orders, exists subquery on lineitem



40

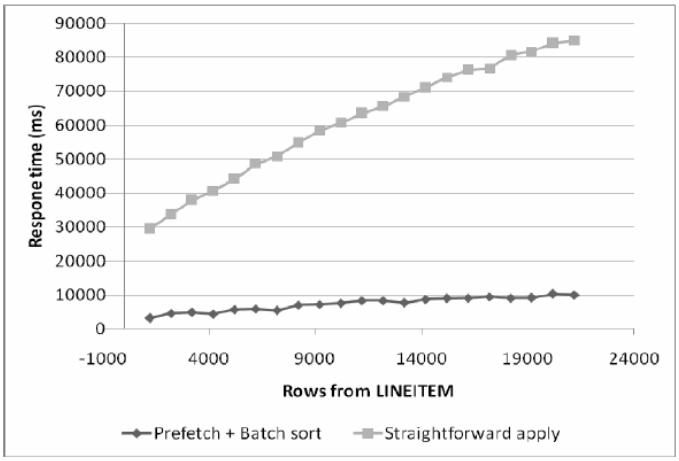
Strategies for Subquery Disjunction

- Section 7.1: One disjunction is a select, other is an exists on a subquery



41

Execution Optimization for Apply



42