Testing the Quality of a Query Optimizer

Michael Stillger
Johann-Christoph Freytag
Institut für Informatik
Humboldt-Universität zu Berlin
Germany
{stillger, freytag}@informatik.hu-berlin.de

Abstract

Today, database technology is used in many different application areas. Therefore, the need to understand how well a particular database management system (DBMS) suits the requirements of a given application has become an important task. One way to address this need is to provide means to measure and to verify the quality of a database management system and its query optimizer. Additionally, since database implementors continue to improve the query optimizer of their specific systems, it becomes especially important for them and the users of those systems to evaluate those changes on a large scale. In particular, changes of the optimizer must be understood by both groups of people. Individual test environments or standardized benchmarks are commonly used to evaluate the quality of an optimizer. Almost all of them are only suited for a particular, artificial database schema and lack the flexibility of determining the size and the shape of queries to be tested and the database to be used.

We present a set of tools which are designed to overcome these problems. As a result it is especially useful for testing query optimization issues like join order, selectivity estimation and choice of execution algorithms. The main features are: specification and generation of the data and of the database schema, specification and generation of a particular set of queries for any existing or newly generated database. On the one hand, the tools aim to support the work of the database implementors (DBIs) to design their own testbed according to changes or enhancements done; on the other hand, they should help vendors and customers to design individual testbeds that reflect the needs of specific database applications.

With the query generator we already have available a first tool in our tool set. Extensions and additional tools are currently under design and implementation.

1 Introduction

The quality of a query optimizer is influenced by many different parameters. First of all, the search strategy used, the quality of the cost functions and the repertoire of the transforma-

tion strategies heavily determine the quality of the final query evaluation plan (QEP). It also depends on the overall optimization time that is available to the optimizer. In many cases it is alsoimportant to the users of a DBMS to rely on a stable and predictable behavior of the optimizer. This is especially true after changes done by DBIs to enhance the optimizer component of the database. Often, these changes might be beneficial for some queries, but it also might impact the optimization of other queries adversely. That is, during the development cycle of the database software, the optimizer might undergo important modifications. The DBI must prove that those changes improve the quality of the QEPs generated. For example, The DBI might decide to add new transformation rules to a rule based query optimizer [DG87], [PHH92] or to redesign the optimizer completely when working with an optimizer generator tool [BMG93],[GM93]. Exchanging the search strategies in the context of large join queries is also an important change whose impact must be verified and checked [LV91]. Thus, a DBI who must go beyond the verification of the changes analytically, must have tools to build various test databases and test queries. Our generator tools aim to be an important utility in this context.

Another, more standardized testing methodology is the use of benchmarks. A benchmarks test suite allows a user to compare different DBMSs independent of the claims made by the database vendors. The Benchmark Handbook by J. Gray provides a good overview of the benchmark methodologies that have been developed so far [Gra91]. The currently existing benchmarks, however, are limited in the number of queries tested and in the number of schemas used. The WISCONSIN benchmark consists of a set of 32 queries that run on a predefined schema [BDT83]. The Set Query Benchmark is designed for a particular area in database systems. Queries with multiple selection predicates run on a large customer database where joins are not needed [O'N91]. Hence, the generated database is restricted to one large relation and the chosen queries reflect the needs of business applications. The most flexible benchmark is the AS3AP [BOT91] benchmark. A unique feature is its adjustable database size which is adapted to the system architecture in such a way that the benchmark can run in a 12 hours limit. But the query set is still limited in range and number.

Our set of tool consists of two parts; a query generator that creates a set of queries which can be run against any existing database, and a database generator that creates the schema and the data according to a given specification. This provides full flexibility for designing and generating individual testbeds. The query generator enables the user to create queries that stress a very specific optimization task on a given database.

Example 1: A new join algorithm that creates a temporary index for each attribute without index is linked into the system. A test set may consist of 30 queries with following properties each:

- a 3 way join
- each join has at least one attribute without index
- one of two relations have a minimum cardinality of 100.000 rows.
- one join attribute is subject of an ORDER BY clause

The 3 joins with at least one non-indexed attribute makes the new join method an applicable execution algorithm in the QEP. Assuming that creating a temporary index is an expensive

operation the algorithm might only be considered when large table joins are performed (> 100.000). The temporary index is also useful to speedup the sort operation of the ORDER BY clause. Hence, these properties ensure that the optimizer considers the new join algorithm in its QEP search. **(end of example)**

Furthermore, it might be likely that the DBI wants to submit queries to a specific database with a specific schema to test certain new or changed features. It is therefore necessary to create "artificial" databases to test those features during query optimization. This database can be created with the database generator (See Figure 1).

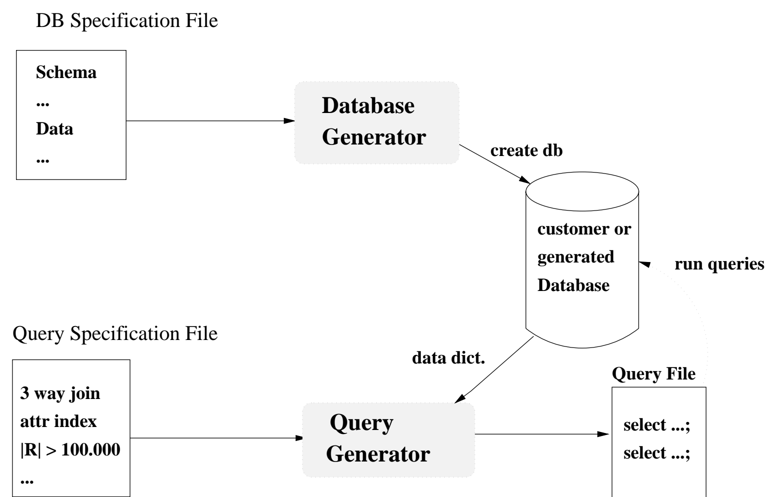

Figure 1: Overview

The data in the database is also crucial for optimization. Unusual selectivity factors for joins and selection predicates, and skews in the value distribution might heavily impact the quality of the QEP generated. The database generator allows the DBI to specify properties of the data stored which he believes are relevant for the optimization process.

2 The Query Generator

This chapter provides an conceptual overview of the query generator and the features that are met.

2.1 Schema independence

Schema independence is the ability to attach to any database and to produce valid queries. We achieve this goal by scanning the data dictionary prior to generating any queries. The generator needs information about the relations, attributes and data types of a particular database. In addition to the structure of the schema, the program collects the available statistics and index information from the data dictionary. Thus, the generator stays completely independent from a predefined database structure as used in the common benchmarks.

2.2 Specification language

Since we cannot predict the kind of queries that are needed for the different tests of the optimizer, it is desirable to cover the complete spectrum of the SQL standard. The generator must be able to produce any query from the infinite set of possible queries that are applicable to a given schema. We designed a query description language that helps the user to specify any range or subclass in which the queries should be generated. The language consists of "meta-SQL" clauses which describe more than the syntactical features of the query to be generated. We assumed that properties of the underlying database should influence the generation of queries. It was therefore necessary to incorporate also physical properties of the data that are beyond the syntactical constructs of the SQL language. For example, the generator can randomly determine an attribute possibly in a SELECT clause or in a selection (join) predicate in the WHERE clause in case no specific attribute is given. In this case, it should be possible to restrict this attribute by a data type specification or by specifying that it should be an indexed attribute. Regarding relations in the FROM clause, it is sometimes more suitable to define the participating relations by their cardinality rather than by name.

Example 2: The following example shows a simple generation script for our generator:

```
Queryname : Index_Scan;
Selectclause :
        Column_numbers = 1 ;
        Column_names   : ANY ;
        Agg_numbers    = No ;
        Keywords       : ORDER BY ANY;
        Indices        =  No ;

Fromclause   :
        Tablenumbers   = 1 ;
        Tables:          ANY ;
        Cardinality_list: 50.000 : 100.000

Whereclause :
Single_Predicate: Number_one
        {
        Operator       : < ;
        Column         : ANY ;
        Indices        = No  ;
        }
```

This specification determines that the SELECT clause can contain any one attribute. There are no aggregate operators and the ORDER BY clause is possible on a non-index attribute. The FROM clause contains one table whose size is between 50,000 and 100,000 tuples. The WHERE clause consists of one simple selection predicate which compares any non-indexed attribute with a constant using the $<$ operator. The specification generates a single table query that scans

an arbitrary relation with the specified size and applies a selection predicate on a non index attribute. The result is then ordered on the attribute specified in the ORDER BY clause.

```
SELECT job_id
FROM employee
WHERE salary < 60.000
ORDER BY job_id
```

(end of example)

In the current version of our generator we can specify the following properties:

- **SELECT clause of SQL query**
  - number of columns (range) in the select clause,
  - specific columns that should always appear in each query,
  - number of columns (range) with primary keys,
  - number of columns (range) with secondary key,
  - number of aggregate columns (range), possibly specific ones, including data type specifications of columns that should be used.
- **FROM clause of SQL query**
  - number of tables (range), possibly specific ones,
  - tables of certain size (cardinality)
- **WHERE clause of SQL query**
  - single predicate, possibly with a specific column name with or without index, with a specific operator, and a specific constant,
  - multiple predicates consisting of single predicates, possibly composed by a specific and/or structure,
  - single join predicate, possibly with specific column names, with number of columns (range) with primary or secondary index,
  - multiple join predicate,
  - complex predicate consisting of join predicates and single predicates, possibly composed by a specific and/or structure.
- **Aggregation clauses of SQL query**
  - number of aggregates,
  - specific aggregate functions,
  - specific types on which to generate aggregate functions.
- **GROUP BY clause**
  - number and type of columns according to the dependencies with the SELECT clause.

2.3 Realistic queries

Before a new query is written to the output file it is submitted to the DBMS for precompilation. This ensures that every query is valid for an automatic evaluation tool that uses the query file as input. Syntactical correctness is a necessary but not a sufficient property for a realistic test set. Consider a join predicate which was created based on the same data type shared by both attributes: *(... WHERE room.number = employee.age)*. Such an semantically incorrect join predicate should never be generated. We solve the problem of generating semantically correct join queries by choosing join predicates from a predefined set of attribute pairs. These pairs are stored as "feasible joins" as part of an extended data dictionary. Based on our experience we strongly believe that this additional input improves the usability of this tool by avoiding queries that do not make sense from the user's point of view.

Similarly, the problem of generating constants for selection predicates in the WHERE clause *(... WHERE employee.age > 332)* need similar attention. Again, we must avoid meaningless constants (unless specified explicitly by the user). For this purpose we use statistical information which is either provided by the DBMS or which must be collected (as in the case of RDB/VMS) in the extended data dictionary prior to the generation of queries (figure 2).

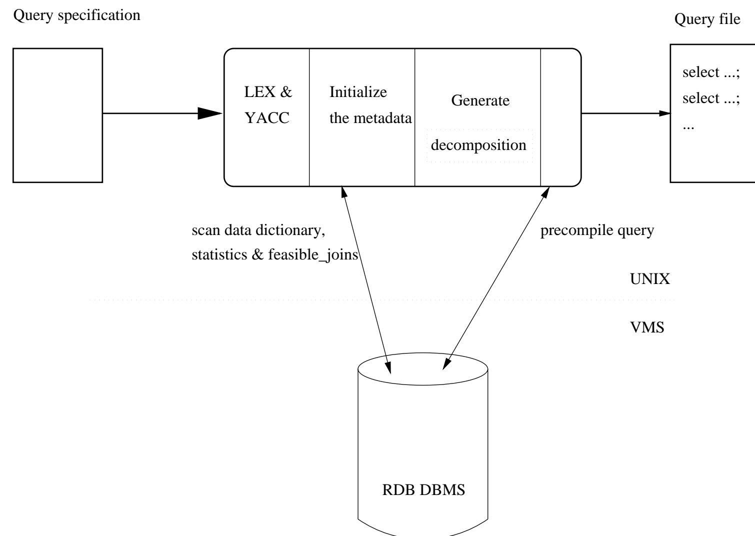

Figure 2: the query generator

2.4 Query decomposition

In many cases , the efficiency of the QEP generated is not the only feature that determines the quality of the optimizer. Often the question if the optimizer generates **correct** QEPs is of much more importance especially for complex queries. We extended our generator such that a query generated is decomposed into a set of "simpler queries" which the optimizer handles correctly.

These queries are embedded into a set of "insert into temp_relation" statements. Together with the matching "create temp_relation" statement each of the query materializes the interme-

diate results in the database. Again, we believe that the execution of simpler queries is more likely to produce a correct result than the original query. Thus, this feature is a useful add-on to verify the correctness of the optimizer, i.e. the correctness of the QEPs generated.

2.5 Implementation

In a first phase, we implemented the query generator for RDB/VMS and Transbase (a database system built at the Technical University of Munich). We used the generator writing many different scripts generating hundreds of queries. Those were run against databases stored in both DBMSs. To make this tool available on a more popular platform we are currently porting the generator to Sybase.

3 The Database Generator

The database generator provides a useful tool to generate arbitrary schemas with arbitrary extensions. Thus a database can easily be generated and filled with user-defined data. This might even better match the particular needs for testing a specific feature of the optimizer than a real database.

The database generator provides a graphical editor (GraphEd) as a front end user interface. The user can select a schema from several classes and edit its graphical representation (figure 3). Nodes represent relations while edges reflect relationships between them, based on possible join predicates. Each node contains a description of the attributes and its data. The database

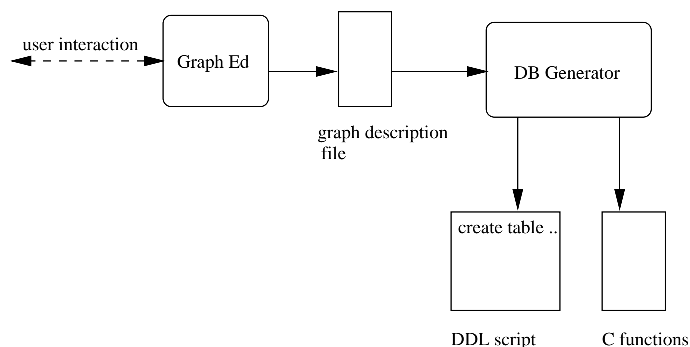

Figure 3: Database Generator

generator is still under development (also for Sybase).

3.1 Schema classification

Determining the join order is an important step during query optimization. We must therefore be able to generate large database schemas that allow the user to generate queries with many joins. Thus, the queries need a minimum number of relations that can be joined. We can classify join queries into star, chain, cycle, clique, grid and linked double chain according to [SMK93].

A DBI must be able to generate different database schemas according to the kind of queries that should be generated. Therefore, a database schema can be represented as a graph similar to the join graphs (see figure 4).

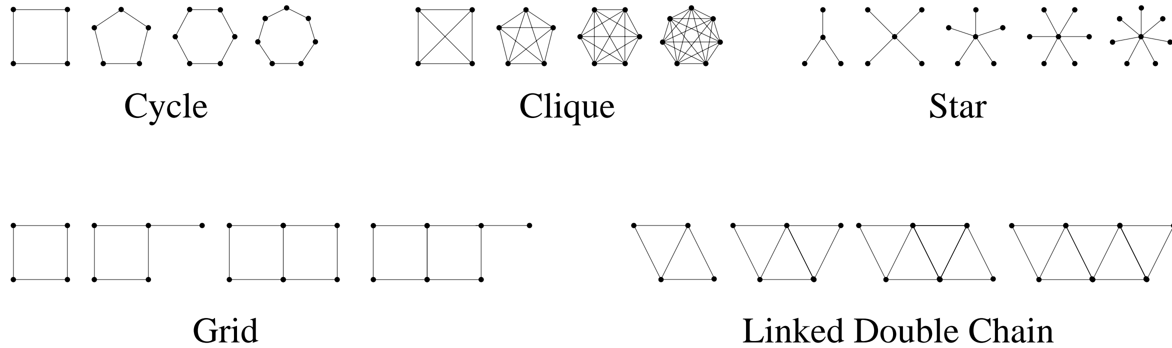

Figure 4: graph classes

3.2 Schema manipulation

The user can also edit the relational schemas and their relationships with a graphical editor. For example, he can add additional relations or can define new join edges. A new edge (relationship) between to relations can lead to additional attributes for joining relations (for example if a “join relationship” cannot be created otherwise). The user can also edit the properties of the nodes representing relations. New attributes can be added, existing one can be changed with regards to their types or domains. Once the user finishes the design, the graphical schemas can be exported for later use.

3.3 Data description

Selectivity estimations are crucial for predicting the correct size of intermediate results. The cost functions of the optimizer must be able to deal with any kind of skewed data. To test the optimizer based on different data distributions, the database generator includes additional parameters to specify some data properties for the extensions of different relations. Each attribute of the node can includes a description of its data. The user should be able to select among several distribution functions (normal, gauss or Zipf), chose duplicate factors and domain ranges (minimum and maximum values).

4 Summary

The database generator and the query generator complement each other. They both allow DBIs to build their own test suites in a simple and straightforward manner to measure and to verify the quality of a given database management system. We also believe that they together will speed up the test phase when changes in the optimizer's strategy occur. For example, new

transformation rules might generate good QEPs for one kind of queries, but generates worse QEPs for other queries than with the previous set of rules.

With our second tool, we extend the capabilities to build powerful test suites. DBIs and users can generate specific databases schemas and a specific database contents. This tool is currently under design and implementation.

Our vision is to design a third tool for our testing environment. This tool should allow us to evaluate (semi-automatically) the measurements that have been collected from the execution of the queries generated. Only with this additional tool, we then believe that we provide a complete testbed for evaluating the quality of a query optimizer.

References

[BDT83] D. Bitton, C.J. DeWitt, and C. Turbyfill. Benchmarking Database Systems: a Systematic Approach. In *Proceedings Very Large Database Systems (VLDB) Conference*, pages 8–19, November 1983.

[BMG93] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *Proc. ACM SIGMOD Conf.*, page 287, Washington, DC, May 1993.

[BOT91] D. Bitton, C. Orji, and C. Turbyfill. The AS3AP benchmark. In J. Gray, editor, *Database and Transaction Processing Sys. Performance Handbook*. Morgan Kaufmann, San Mateo, CA, 1991.

[DG87] D. DeWitt and G. Graefe. The EXODUS Optimizer Generator. In *19 ACM SIGMOD Conf. on the Management of Data*, May 1987.

[GM93] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 209, Vienna, Austria, April 1993.

[Gra91] J. Gray, editor. *The Benchmark Handbook*. Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1991.

[LV91] R.S.G. Lanzelotte and P. Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proceedings of the 17'th VLDB Conference*, 1991.

[O'N91] P.E. O'Neil. The Set Query Benchmark. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann Publishers, Inc. San Mateo, CA, 1991.

[PHH92] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1992.

[SMK93] M. Steinbrunn, G. Moerkotte, and A. Kemper. Optimizing Join Orders. Technical report, Universitaet Passau, Germany, 1993.