Bulletin of the Technical Committee on

Data Engineering

September, 1995 Vol. 18 No. 3 **IEEE Computer Society**

Letters

Special Issue on Database Query Processing

Conference and Journal Notices

Letter from the Editor-in-Chief

About this issue

Query processing has been, and will continue to be, enormously important to the successful application of databases to real user problems. Indeed, users are placing increased demands on databases in such areas as data warehousing, on-line analytic processing (OLAP), distribution, etc. In such applications, databases may be multi-gigabytes, even terabytes, in size. And the queries can become very complex, involving multiple joins and multiple aggregations. It is a crucial and challenging task to ensure that such queries can execute efficiently. The current issue was assembled by Goetz Graefe, a leader in the field both of query processing technology and its application to real database systems. It includes papers solicited by Goetz on some of the current active areas in query processing and its application, including how one might assess the job done by an optimizer. I want to thank Goetz for taking time from his very hectic Microsoft schedule to put the issue together. I think you will find it rewarding reading.

State of the Bulletin

As I indicated in my last letter (June issue), while we have not resolved our long term financial situation, we continue to enjoy the support of the IEEE Computer Society and its Technical Activities Board (TAB). This support, for which we are grateful, is manifest in the TAB's providing us with sufficient budget to continue, on a year to year basis, with the hardcover publication and distribution of the Bulletin.

We continue to make the Bulletin available electronically via the Microsoft FTP server. The procedure for this is to log on to

```
ftp.research.microsoft.com
```

as "anonymous" and give as your password your email address. Change directories as follows-

```
cd  pub
cd debull
```

You can then do a "dir" or "ls" to determine what files are present. Read the README file in the debull directory for more complete information.

Having selected the issue you wish, and the format you desire, select the appropriate file with the ftp "get" command, e.g.

```
get sept95-letfinal.ps
```

to have the file delivered to your system. The files are all in postscript so you will need a postscript viewer and/or printer to be able to read them. *Note that the files now come with the file type ".ps".*

We are in the process of setting up a home page to make the Bulletin available on the web. More on that will be forthcoming shortly.

David Lomet
Editor-in-Chief

Letter from the Special Issue Editor

This special issue of the bulletin is about query optimization. While relational query processing, both optimization and execution, might feel like a ”done deal” to many researchers, some of us disagree. There certainly are lots of open issues and questions within special topics such as on-line analytical processing (OLAP), multi-dimensional analysis, data warehousing, data mining, parallelism, resource optimization and management, recursion, object-relational queries, and distributed query processing; the latter in particular over loosely connected networks such as the internet and ”intra-nets” within individual organizations. However, there are also a number of very fundamental issues that pertain to any relational database system in just about any application. In the present issue, we have a collection of papers that address several Achilles heels (if a number greater than two is permitted here) of most commercial relational database systems.

In the first paper, Chaudhuri and Shim address optimization of complex SQL queries with aggregations over multiple tables. While the transformations discussed here (and by Yan and Larson in their work) may seem relatively straightforward in ”pure” relational algebra, the semantics of SQL with empty tables, NULL values, and duplicate rows require very thorough consideration.

The second paper, by Ioannidis and Poosala, summarizes their work on histograms, where the notion of histograms is extended to include grouping by range (the traditional notion of histograms) as well as grouping by frequency – in the extreme case, each one among the most frequent values for a table’s column forms its own histogram bucket, a technique used in some IBM database products.

The third paper presents yet another extensible framework for database query optimization, which was used to implement a special search strategy for complex relational queries, as described in the fourth paper, by Ozkan, Nural, Koksal, Altinel, and Dogac. It is true that extensibility is often achieved with a certain loss of efficiency, and an extensible framework for database query optimization must include a wide variety of hooks for guidance, pruning, etc.

The fifth paper describes one piece in the development of commercial query optimizers that is usually forgotten in the research literature, quality assurance. While all database vendors have batteries of database schemas and queries to ensure that the query processor produces the same result after a modification, systematic and time-efficient testing of a query optimizer’s effectiveness, i.e., the quality of its plans, is often neglected. Stillger and Freytag describe a query generator that can be used to stress-test an optimizer, and that will be tremendously useful once coupled with automated analysis tools.

There are great challenges ahead for researchers and practitioners in database query optimization and execution. Having left my academic career and working on industrial products now, I have started to gain an appreciation for the myriad of difficulties and concerns that I can no longer simply ”define away.” I hope that the present collection of papers, while all focused on well-known problems, may be considered stepping stones towards better database products.

Goetz Graefe
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

An Overview of Cost-based Optimization of Queries with Aggregates

Surajit Chaudhuri
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
chaudhuri@hpl.hp.com

Kyuseok Shim
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
kshim@almaden.ibm.com

1 Introduction

The optimization problem of Select-Project-Join queries has been studied extensively. However, a problem that has until recently received relatively less attention is that of optimizing queries with aggregates. For example, for single block SQL, traditional query processing systems directly implement SQL semantics and defer execution of grouping until all joins in the FROM and WHERE clauses have been executed. Furthermore, for queries that reference views with aggregates, traditional optimizers do not consider flattening such views. In this paper, we will show that there is a rich set of execution alternatives that can significantly enhance the quality of the plans produced. We also discuss how one can choose among the alternatives.

First, let us consider single-block SQL queries. For these queries, the reason to consider alternatives where a group-by operation precedes a join is that an early evaluation of the group-by can significantly reduce the size of the input to the join. Such a transformation may also result in the grouping operation to be pushed down to a base table. In such an event, one can use indexes on the base tables to combine the operation of the join and the group-by.

Next, in case of queries containing views with aggregates, the presence of Group By hinders the ability to reorder relations within the view with relations outside. Yet, reordering joins across view boundaries has the potential of generating cheaper execution alternatives.

In order to be able to optimize queries with aggregates, we need:

- Transformations that enable us to optimize queries containing Group By and join.
- Optimization algorithms that can incorporate the above transformations without increasing the search space dramatically.

We begin by considering optimization of single-block SQL queries. We refer the readers to [CS94, CS95, CS96] for the technical details.

2 Transformations for Single Block SQL

There are several transformations that make it possible for a group-by to precede a join. The conditions that make such transformations possible fall in the following broad categories:

(1) Conditions that depend solely on the schema and the nature of the join predicates and independent of the nature of the aggregating functions.

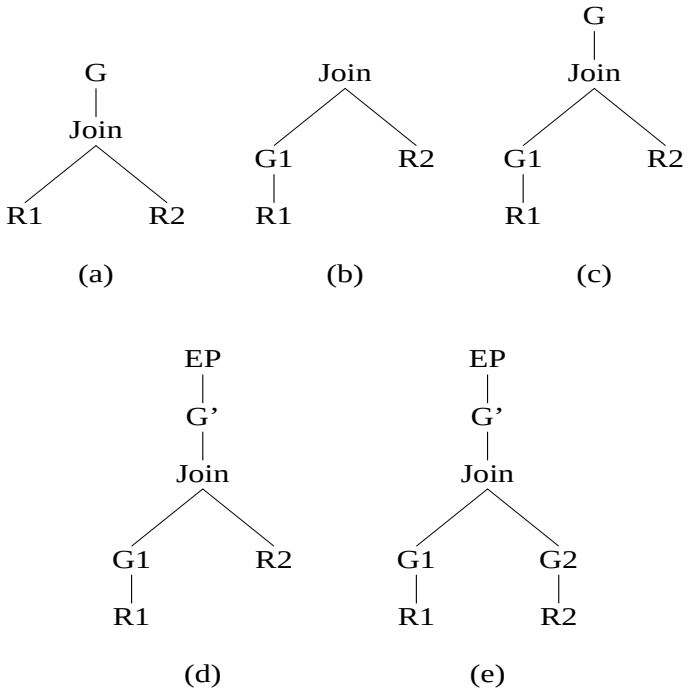

Figure 1: Transformations

(2) Conditions that depend on specific properties of aggregating functions. This can lead to *introduction* of additional group-by nodes.

(a) Conditions that do *not* require introducing derived columns.

(b) Conditions that require use of *derived columns*

Transformations that belong to group (1) are applicable for any aggregating function, including any side-effect free *user-defined function*. An application of a transformation to the operator tree in Figure 1(a) leads to the transformed operator tree illustrated by Figure 1(b). Observe that the transformation in the figure does *not* increase the number of operators. For example, consider the case with two tables `Sales(saleid, amount, prodid)`,`Product(prodid, division, state)` and the query is to calculate total sales for each product developed in Western states. In such a case, the traditional execution (Figure 1(a)) takes a join between the two tables and then computes a `Group By` on `prodid`. In the alternative scheme of Figure 1(b), we first group by `Sales` table and compute the total sales for each product. Subsequently, we eliminate results that do not correspond to products developed in the western states. If the selectivity of the join is large, then the execution of Figure (b) may be more efficient.

By exploiting the properties of the aggregating functions make it possible to do "stagewise" group-by leading to *introduction* of the group-by nodes (e.g., in Figure 1(c)). These aggregation functions follow the property:

$$Agg(S \cup S') \quad = \quad Agg(Agg(S) \cup Agg(S')) \tag{1}$$

where S and S' are bags of values and $\cup$ denotes SQL's union-all operation on the bags. For example, the aggregate function SUM follows the above property since $SUM(S \cup S') = SUM(SUM(S) \cup SUM(S'))$. Let us consider a simple modification of the query given earlier. Assume that we want to compute the total sales for each division where a division may have many products. By applying the transformation in Figure 1(c), we can compute the total sales for each product, then join with `Product` table, and finally add up the sum of sales for products that belong to the same division.

Finally, there are cases where keeping track of the *count* of the coalesced groups makes it possible to introduce group-by or to do an early group-by. In such cases, the properties of the aggregating functions that may be admitted can be more general than Equation 1:

$$Agg(S \cup S') = f(Agg_1(S), Agg_1(S'), Count(S), Count(S')) \quad (2)$$

where S and S' are bags of values and $\cup$ denotes SQL's union-all operation. However, computing stagewise grouping now requires use of derived column values as well. For example, given the count and the average for two groups, the average of the coalesced group can be computed. This is illustrated by Figure 1(d) where the node EP (*extended projection*) represents the computation of the aggregated value using the relationship in Equation 2. Observe that the execution of EP and the join node can be combined in one single join implementation.

Transformations (b)-(d) represent early evaluation of grouping on only one operand of the join node. They are generalized in Figure (e) to both sides of the operands and represent aggressive use of early grouping.

3 Optimization Algorithm for Single Block Query

The decision whether or not to apply a transformation needs to be made by the optimizer based on anticipated costs. However, such choice also *interacts with join ordering* since group-by has the effect of reducing the number of tuples at the cost of requiring sorting or hashing (unless the data stream is pre-sorted). Thus, the problem of optimization in the presence of group-by and aggregates has the following properties:

1. *Increase in the number of operators:* A group-by operator may be added, i.e., the number of operators may increase, e.g., transformation in Figure 1(c),(d)&(e).

2. *Effect on the Physical Properties:* Application of a grouping operation G on a relation R results in a relation not just with fewer tuples than in R, but also with differences in the physical properties:

   (a) Application of a group-by operator may *change* the width of the intermediate relation.

   (b) Application of a group-by operator interacts with *interesting orders*[SAC$^+$79].

The most difficult problem in handling the optimization problem in the presence of group-by arises from the fact that the transformations can introduce *additional* group-by operators. Thus, the potential execution space is dramatically increased since there may be an opportunity to place a group-by preceding every join!

Another challenge in optimizing queries is that it is not possible to locally compare segments of plans such as in Figure 1(a) and Figure 1(b) and to choose the winner, for two reasons. First, in grouping early, a new interesting order may be introduced if the grouping is implemented by sorting. Thus, although Figure 1(b) may not win locally, its order generated by sorting may reduce the cost of future join operations. This effect is similar to that of considering a sub-optimal sort-merge join in traditional Select-Project-Join queries. Moreover, one needs to address the problem of determining the major to minor order for sorting (if the grouping is implemented by sorting) and its relationship to the join predicates. Similar considerations arise if grouping is implemented by hashing as well. Next, observe that if there are more than one aggregation functions on the same column, then the width of the relation after application of the grouping will increase. The effect of increased width leads to an effect converse to that of interesting order. Thus, even if Figure 1(b) was locally better than Figure 1(a), it may perform suboptimally because increased width may lead to higher cost globally.

3.1 Greedy Conservative Heuristic

We proposed *greedy conservative* heuristic as a technique to optimize single block SQL with group-by [CS94] by exploiting the transformations in Figure 1. The design principles of greedy conservative heuristic were to ensure

that: (1) the quality of the plan produced is no worse than the execution plan produced by a traditional optimizer and (2) there is no significant overhead in optimization.

In order to ensure that the cost of increase in optimization is modest, we decided that the choice between applying an early grouping and late grouping must be made *locally*. The greedy conservative heuristic places a group-by preceding a join *if and only if* the following conditions hold:

(1) It is semantically correct.

(2) The record width in the output of the group-by operator is no larger than the one in its input.

(3) It results in a cheaper plan for *that* join.

Thus, while considering the join between two relations R and S, if the above conditions (1) and (2) hold, the cost of the optimal plan for $R \bowtie S$ is compared against the cost for $G(R) \bowtie S$, where $G(R)$ represents application of a group-by. Note that either R or S (or, both) can be intermediate relations. Furthermore, the greedy conservative heuristic considers only a *single* ordering of the group-by columns for each join method considered for implementing $R \bowtie S$. Thus, when sort-merge is used for the join method, we choose a major to minor ordering that is the same as that for the join node. Thus, by virtue of the above local decisions, the greedy conservative heuristic adds little overhead in optimization while considering a significantly extended execution space.

As pointed out in the previous section, a local decision such as those made by greedy conservative can be suboptimal. However, the greedy conservative heuristic ensures that the local decision does not result in a plan that is worse than the traditional execution. To enforce such a property, a plan such as $G(R) \bowtie S$ is considered only when the width of the intermediate obtained by $G(R)$ does not increase (i.e. Condition (2)). Our experiments with the implementation (an extension of [SAC+79]) indicate a significant increase (sometimes an order of magnitude) in the quality of the plans for a very modest increase in the optimization overhead [CS94]. Our results are particularly encouraging in view of the fact that we took a conservative cost model that discourages early evaluation of group-by operators. Although our implementation was with a System-R style optimizer, the greedy conservative heuristic can be incorporated in other optimizer architectures as well.

4 Beyond Single Block Queries

In general, a query that uses a view definition with aggregate, cannot be turned into a single-block Select-Project-Join (SPJ) query. In this section, we will discuss how group-by operators can be moved in the traditional query graph across query blocks. Relative placement of group-by across query blocks has significant impact on the cost of the plan of a query (with views containing aggregates) due to its effect on data reduction and reordering of relations across query blocks. In this section, we discuss a new approach to optimizing such queries.

Pull-up Transformation

We can use a transformation to pull up the group-by operator above join. Such a transformation is interesting since by pulling up the group by, it enables alternative join orders. Indeed, a sequence of pull-up transformations may result in collapsing a multi-block query into a single-block query. In this section, we explore how *given an aggregate view*, we can obtain a query with group-by on the top.

Let us consider the execution plans `P1` and `P2` in Figure 2. In the figure, `R1` and `R2`, `J1` and `J2`, and `G1` and `G2` are base tables (or query expressions), join nodes and group by operators, respectively. In this plan, we defer the group-by to obtain an equivalent plan `P2` in the following manner (See Figure 2): The grouping columns of `G2` will be the union of the grouping columns of `G1` and the key (columns) of `R2`. The intuition is that by "tagging" each tuple of `R1` with the key of `R2`, we can recover the original groups by including the key attribute of `R2`

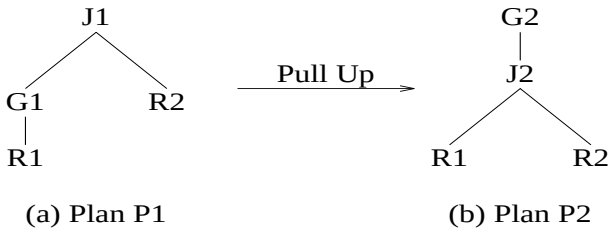

Figure 2: Pulling Up a Group-By operator

among the grouping columns of `G2`. Subsequent to deferred grouping (`G2`), we can throw away the key column of (`R2`) unless it is part of the projection columns. Note that since in `P2`, `G2` has key column(s) of `R2` among its grouping columns, the projection column of the join node `J2` includes these columns as well. Furthermore, the projection list of the join node `J2` includes aggregating columns. The transformation is applicable only if the join predicate in plan `P1` do not involve any aggregation column of `G1`. The formal statement of the transformation appears in [CS96].

The paper by Dayal [Day87] briefly touches on the idea of using keys for pull-up. Ganski-Wong [GW87] implicitly used a restricted form of pull-up for the specific class of Join-Aggregate queries that they optimized. These are special instances of the pull-up transformations defined above.

Optimization Algorithm

The class of queries $\mathcal{Q}$ for which we consider the optimization algorithm has the canonical form shown in Figure 3 (See [CS96] for details):

- Every aggregate view V_i consists of a select-project-join expression with a Group By clause and possibly a `Having` clause. Thus V_i contains a join and a group-by operator G_i.
- The query is a single block query, consisting of a join among aggregate views and a set of base tables i.e., join among $V_1, ..V_m, B_1, ..B_n$ where V_i are aggregate views and B_i are base relations. The query may contain a Group By and a `Having` clause. The set of relations $\{B_1, B_2, ..\}$ and $\{V_1, V_2, ..\}$ are denoted by $\mathcal{B}$ and $\mathcal{V}$ respectively.

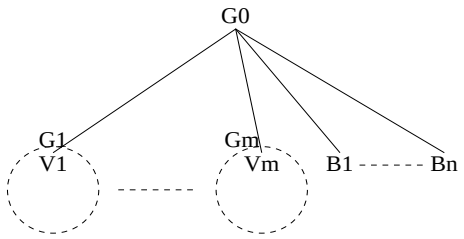

Figure 3: Canonical Form of Query

Our approach to extending the traditional optimization algorithm for multi-block queries by using pull-up as well as push-down transformations is guided by the following two criteria First, our approach should be adaptable to existing optimizers and should produce a plan that never does worse than a traditional optimizer. Next, we want to be able to limit the extent of search in a systematic fashion. Thus, our key idea is to apply pull-up transformation between a view V_i and a relation in $\mathcal{B}$, but not among relations in V_i and V_j where $i \neq j$. Additional reordering among relations in V_i and V_j is enabled by a preprocessing step. The details of such an algorithm is in [CS96].

Relationship to Correlated Nested Queries

We note that the *Join-Aggregate* class of nested queries is closely related to that of optimizing queries containing views with aggregates. This follows from the past work in *flattening* nested queries, pioneered by Kim [Kim82]. For example, the result of Kim's transformation is a query that is a join [1] of base tables and one or more aggregate views, i.e., views containing aggregates. Therefore, the result of optimizing queries containing views with aggregates can be used for optimizing queries with correlated nested subqueries. We can also show that alternative flattening schemes for nested subqueries such as that due to Ganski [GW87] are special cases of as execution alternatives due to optimization of aggregate views. The differences between the transformations due to Kim and Ganski are highlighted by Figure 4 and can be seen as arising due to two extreme placements of the group-by operators. Thus, the problem of optimizing queries containing aggregate views is not only a key problem in its own right, it also directly bears upon the problem of optimizing queries with nested subqueries.

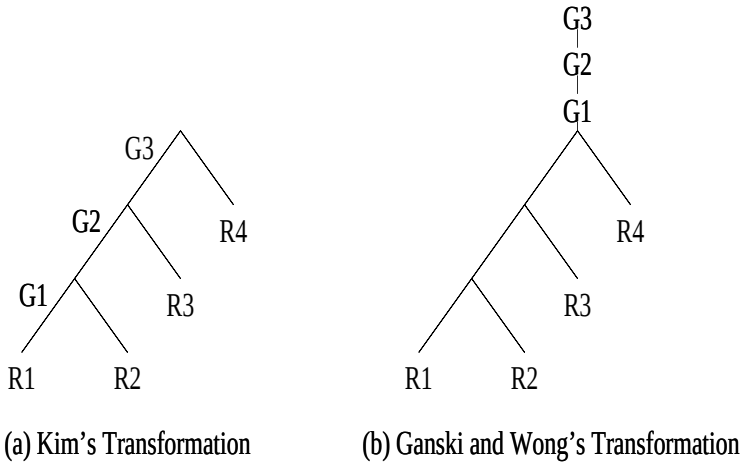

Figure 4: Unnesting Transformations

5 Conclusion

With the increasing emphasis on decision-support systems, the problem of optimizing queries with aggregates have assumed increasing importance. This article has pointed to some of the recent developments in this area. However, the evolving querying models for decision-support systems (e.g., OLAP interfaces) will spur further development in this area.

To Probe Further

Invariant grouping for a single block query was studied independently by us in [CS94] and in [YL94]. Our paper also introduced simple coalescing and generalized coalescing. Later, additional transformations (including pull-up) on group-by were reported in [YL95, GHQ95, CS96]. The problem of optimization using the transformations was addressed by us in [CS94] for single block SQL queries and in [CS96] for multi block SQL queries. A comprehensive treatment of the transformations as well as the optimization algorithms discussed in this article is forthcoming in [CS95].

[1] Note that for correctness, such transformations require outerjoin in the general case [GW87, Mur92].

References

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20th International VLDB Conference*, Santiago, Chile, Sept 1994.

[CS95] S. Chaudhuri and K. Shim. Complex queries: A unified approach. Technical report, Hewlett-Packard Laboratories, Palo Alto, In preparation to submit for publication 1995.

[CS96] S. Chaudhuri and K. Shim. Query optimization with aggregate views. In *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, March 1996.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th International VLDB Conference*, Brighton, August 1987.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the 21st International VLDB Conference*, Zurich, Sept 1995.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, San Francisco, CA, May 1987.

[Kim82] W. Kim. On optimizing an SQL-Like nested query. *ACM TODS*, Sept 1982.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the 18th International VLDB Conference*, Vancouver, Canada, August 1992.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Boston, MA, June 1979.

[YL94] W. P. Yan and P. A. Larson. Performing group-by before join. In *Proceedings of International Conference on Data Engineering*, Houston, Feb 1994.

[YL95] W. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International VLDB Conference*, Zurich, Sept 1995.

Histogram-Based Solutions to Diverse Database Estimation Problems

Yannis Ioannidis[†]
Viswanath Poosala
Computer Sciences Department, University of Wisconsin, Madison, WI 53706
{yannis,poosala}@cs.wisc.edu

Abstract

Many current database systems use some form of histograms to approximate the frequency distribution of values in the attributes of relations and based on them estimate some query result sizes and access plan costs. In this paper, we overview the line of research on histograms that we have followed at the Univ. of Wisconsin. Our goal has been to identify classes of histograms that combine three features in most realistic cases: (i) they produce estimates with small errors, (ii) they are inexpensive to construct, use, and maintain, and (iii) they can be used for many diverse estimation problems. Based on that goal, we present several results, which eventually point towards a class of histograms that are practical, close to optimal, and effective in estimating sizes of query results, frequency distributions of attribute values in query results, and even costs of accesses using secondary indices.

1 Introduction

Database management systems (DBMSs) contain several modules that require estimates of the sizes of query results or of the costs of queries and query access plans. First, the query optimizer needs size estimates so that it can plug them into cost formulas to estimate the costs of access plans. These cost estimates are then used to determine which plan is expected to be the cheapest to execute the corresponding query. Second, a sophisticated user interface needs size and cost estimates to provide them as feedback to users before a query is actually executed, as a way to detect errors in the query or misconceptions about the database.

Both size and cost estimations are usually based on statistics that are maintained in the database catalogs. In many systems (e.g., DB2, Informix, Ingres, Sybase, Microsoft SQL Server), these statistics take the form of histograms, which represent approximations of the frequency distributions of values in attributes of the database relations. These approximations are then used to obtain the necessary estimates.

For the past few years, we have been working on identifying classes of histograms that are desirable for various estimation problems [IC93, Ioa93, IP95]. In this paper, we provide an overview of several earlier and some more recent results, which eventually point towards a class of histograms that are practical, close to optimal, and effective in many estimation problems.

[†]Partially supported by the National Science Foundation under Grants IRI-9113736 and IRI-9157368 (PYI Award), by Lockheed as part of an MDDS contract, and by grants from IBM, DEC, HP, AT&T, Informix, and Oracle

2 Definitions

2.1 Frequency Distributions of Values

Given an attribute of a relation, the *frequency distribution of its values* is a set of pairs indicating, for each value in the attribute's domain, the number of tuples in the relation where the value appears. As an example, consider the simple relation *OLYMPIAN* in Table 1. The frequency distribution of the various values in its Department attribute is shown in Table 2.

OLYMPIAN		
Name	Salary	Department
Apollo	60K	Energy
Aphrodite	60K	Domestic Affairs
Aris	50K	Defense
Artemis	60K	Energy
Athena	70K	Education
Demeter	60K	Agriculture
Ermis	60K	Commerce
Hefestus	50K	Energy
Hera	90K	General Management
Hestia	50K	Domestic Affairs
Poseidon	80K	Defense
Pluto	80K	Justice
Zeus	100K	General Management

Table 1: The OLYMPIAN relation

Department	Frequency
General Management	2
Defense	2
Education	1
Domestic Affairs	2
Agriculture	1
Commerce	1
Justice	1
Energy	3

Table 2: Frequency distribution of Department

One can generalize the above and discuss frequency distributions of combinations of multiple attributes. For example, the joint frequency distribution of the attributes Department and Salary would indicate the numbers of tuples in the relation where each pair of Department-Salary values occurs. Multi-attribute joint frequency distributions essentially capture the correlation between the values of the participating attributes. In principle, such distributions (or their approximations) are required to calculate the size of any query that involves multiple attributes from a single relation. In practice, however, DBMSs deal with frequency distributions of individual attributes only, because considering all possible combinations of attributes is very expensive. This essentially corresponds to what is known as the *attribute value independence assumption*, and although rarely true, it is adopted by all current DBMSs. To simplify presentation, with few necessary exceptions, our description below deals with single-attribute distributions only. Our results, however, hold for the general case.

2.2 Histograms

Below we define histograms in a way that is more general than is common in query optimization. Specifically, a *histogram* of some attribute is an approximation of the frequency distribution of its values obtained as follows: the (attribute value,frequency) pairs of the distribution are partitioned into *buckets*; the frequency of each value in a bucket is approximated by the average of the frequencies of all values in the bucket; and the set of values in a bucket is usually approximated in some compact fashion as well. Note that, although it is uncommon in database histograms, in principle, the ranges of attribute values grouped in two buckets could overlap. Also note that a histogram with a single bucket generates the same approximate frequency for all attribute values. Such a histogram is called *trivial* and corresponds to making the *uniform distribution assumption* over the entire attribute domain.

Department	Histogram H1		Histogram H2		Histogram H3	
	Frequency in Bucket	Approximate Frequency	Frequency in Bucket	Approximate Frequency	Frequency in Bucket	Approximate Frequency
Agriculture	1	1.50	1	1.33	1	1.43
Commerce	1	1.50	1	1.33	1	1.43
Defense	2	1.50	2	1.33	2	1.43
Domestic Affairs	2	1.50	2	2.50	2	1.43
Education	1	1.75	1	1.33	1	1.43
Energy	3	1.75	3	2.50	3	3.00
General Management	2	1.75	2	1.33	2	1.43
Justice	1	1.75	1	1.33	1	1.43

Table 3: Three types of histograms for the Department attribute

Continuing on with the example of the *OLYMPIAN* relation, Table 3 presents three different histograms on the Department attribute, all with two buckets. For each histogram, it first shows which frequencies are grouped in the same bucket by enclosing them in the same shape (box or circle), and then shows the resulting approximate frequencies, i.e., the averages of all frequencies enclosed by identical shapes.

There are various classes of histograms that systems use or researchers have proposed for estimation. Most of the earlier prototypes, and still some of the commercial DBMSs, use trivial histograms, i.e., make the uniform distribution assumption [SAC+79]. That assumption, however, rarely holds in real data and estimates based on it usually have large errors [Chr84, IC91]. Excluding trivial ones, the histograms that are typically used by systems belong to the class of *equi-width* histograms, first discussed for database query optimization in Kooi's thesis [Koo80] (under the name *variable-count* histograms). In such a histogram, there is no overlap among the ranges of attribute values associated with its buckets, and the size of the range (or the number) of values in each bucket is the same, independent of the value frequencies. Since these histograms store much more information than trivial histograms (they typically have 10-20 buckets), their estimations are most often better. Histogram H1 above is equi-width, since the first bucket contains four values in the range A-D and the second bucket also contains four values in the range E-J.

Although we are not aware of any system that currently uses histograms in any other class than those mentioned above, several more advanced classes have been proposed and are worth discussing. In Kooi's thesis [Koo80], the reverse of equi-width histograms was also introduced (under the name *variable-range* histograms). In such a histogram, the sum of the frequencies of the attribute values associated with each bucket is the same, independent of the size of the range (or the number) of these values. Piatetsky-Shapiro and Connell gave this class of histograms the name that is currently used most often, *equi-depth* (or *equi-height*). Their extensive study showed that equi-depth histograms have a much lower worst-case and average error for a variety of selection queries than equi-width histograms [PSC84]. Muralikrishna and DeWitt [MD88] extended the above work for multidimensional histograms that are appropriate for multi-attribute selection queries. As mentioned above, our line of work has focused on identifying optimal histograms (those with the least error in their estimates) and has introduced serial and end-biased histograms, discussed in detail below.

2.3 Serial and End-biased Histograms

For many types of queries, an important role with respect to optimality is played by the class of *serial* histograms [IC93]. In those, the frequencies of the attribute values associated with each bucket are either all greater or all less than the frequencies of the attribute values associated with any other bucket. That is, the buckets of a serial histogram group frequencies that are close to each other with no interleaving. Histogram H1 in Table 3 is not serial as frequencies 1 and 3 appear in one bucket and frequency 2 appears in the other, while histograms H2 and H3 are.

An important subclass of serial histograms is that of *end-biased* histograms. In those, some number of the highest frequencies and some number of the lowest frequencies in an attribute are explicitly and accurately maintained in separate individual buckets, and the remaining (middle) frequencies are all approximated together in a single bucket. End-biased histograms are serial since their buckets group frequencies with no interleaving. Histogram H3 in Table 3 is end-biased, since it accurately maintains the frequency of the Energy Department, while averaging the frequencies of all others. One could define an intermediate class of histograms as well (subclass of serial and superclass of end-biased), where the middle frequencies are partitioned into multiple buckets. These histograms, however, have all the problems of serial histograms and not all of their advantages (both to be discussed later), and are therefore not interesting.

3 Histogram Effectiveness for Various Estimation Problems

In this section, we focus on four estimation problems. Each of the following subsections deals with one of these problems, defines a desirable notion of optimality with respect to estimation errors, and then describes our results regarding optimal or suboptimal but effective classes of histograms. All results assume that histograms maintain the precise set of values associated with each bucket. We first introduce some notation regarding a histogram with β buckets:

b_i The i-th bucket in the histogram, $1 \leq i \leq \beta$ (numbering is in no particular order).

n_i The number of attribute values placed in bucket b_i.

V_i The variance of the frequencies of the attribute values placed in bucket b_i.

3.1 Result Sizes of Arbitrary Equality Join and Selection Queries

Assume that for every attribute of every relation, a specific number of buckets has been prespecified for a histogram approximating its frequency distribution. Our goal is, for each attribute of interest of each relation, to obtain a single histogram and use that in all queries where the attribute participates. This histogram may not be optimal in every case, i.e., may not return the closest approximation to the result size of all queries applied on all possible databases, but should be optimal "on the average".

Working with this notion of optimality, our work [IC93, Ioa93, IP95] has shown the following for tree, function-free, equality join and selection queries:

- Among all histograms with β buckets on some attribute, the one that minimizes

$$\sum_{i=1}^{\beta} n_i \, V_i \tag{3}$$

  is optimal.

- Optimal histograms belong to the class of serial histograms.

- Within the restricted class of end-biased histograms, again the one that minimizes (3) is optimal.

3.2 Frequency Distributions of Attribute Values in Results of Equality Join Queries

In addition to the size of a query result, it is often necessary to obtain good estimates for the overall frequency distribution of values in some of its attributes. This is useful in optimization of complex queries, because the distribution of an intermediate result affects the result size and cost of the subsequent operation. It is also useful

in data partitioning for parallel executions of joins, because a good estimate of the join result distribution helps taking into account not only the input relations skew but the result skew as well. By partitioning the input relations so that work is balanced among processors with respect to all skews, response time of a parallel join execution can be reduced.

Working with the same notion of optimality as in Section 3.1, our work has shown the following for single equality join queries:

- The same (serial) histograms that are optimal for query result size estimation (identified in Section 3.1) are also optimal for the estimation of the frequency distributions in any attribute of these query results.

Thus, a single histogram can achieve optimality on multiple fronts.

3.3 Result Sizes of Range Selection Queries

Unlike equality selection predicates, the result size of a range predicate depends on the frequencies of the attribute values as well as the distribution of the values in the attribute's domain. Based on our work, as well as on some results from numerical analysis [dB78], we believe that a formal characterization of optimal histograms is unlikely for this problem. Hence, we have focused mainly on the design and efficient construction of highly accurate, but not necessarily optimal, histograms. This work is performed in collaboration with researchers from IBM Almaden and so far has resulted in the following:

- Even if not optimal, the (serial) histograms that are optimal for the previous two estimation problems are rather effective in estimating the result sizes of range queries as well. Although one would expect that histograms grouping arbitrary attribute values in buckets would not perform well for range queries, the fact that serial histograms approximate frequencies very well makes them reasonably accurate for these queries as well.

- A new class of histograms that accurately maintain the high frequencies in the data is also very effective. These histograms are similar to end-biased histograms, but permit more than one bucket to group multiple different frequencies so that each bucket covers a contiguous range of attribute values. They perform particularly well for skewed distributions, which usually cause the highest errors in estimation.

For range selection queries, both types of histograms significantly improve upon the traditional equi-width and equi-depth histograms as well as other statistical approximation techniques (e.g., [JC85]). Which of the two is to be preferred is an issue that we are currently investigating experimentally.

3.4 I/O Costs of Relation Accesses through Secondary Indices

All the preceding estimation problems require histograms on frequency distributions of attribute values. As mentioned earlier, in their general form, some of these problems deal with multi-dimensional (multi-attribute) frequency distributions, which capture the *logical* correlation between the values of the participating attributes. There is nothing to prevent us, however, from making one of the dimensions to be the pages of a relation, thus capturing the *physical* correlation of the values in the remaining attribute(s) to these pages. For example, for the two-dimensional case, we can talk about frequency distributions that represent the number of tuples with a specific attribute value stored in a specific page, for all such value-page pairs. In other words, such distributions capture the level of clustering of the data on disk, and can be used (in accurate or approximate form) in many ways. One of their most interesting uses is in calculating the number of data pages accessed when scanning a relation or processing a selection using an unclustered index. This depends heavily on the level of data clustering on disk, as the latter (together with the buffer pool size) determines whether or not a page will be fetched more than once.

Given the fact that physical correlation distributions cannot be maintained accurately, we propose that they are approximated by histograms, as done for logical correlation distributions. General questions on histogram optimality and effectiveness are still part of our investigation. With respect to equality selection using a secondary index, however, one can show that cost calculation for such plans using physical correlation distributions is isomorphic to query result size calculation using logical correlation distributions. This observation indicates the potential of our approach.

4 Histogram Implementation

Based on the above analysis, the importance of serial histograms (and their subclass of end-biased histograms) for several estimation problems becomes evident. Therefore, we turn our attention to how such histograms can be implemented. In particular, there are four issues that we briefly discuss: how the relevant data is collected from the database; how the buckets of the histogram are identified; how the histogram is stored in the database; and how the histogram is used for estimation. The following discussion addresses only histograms of logical frequency distributions. The exact same arguments, however, apply to histograms of physical frequency distributions as well, which are useful for cost estimations (Section 3.4).

4.1 Data Collection

In principle, to construct a histogram on some attribute R.A, one has to first execute the query

select A, Count (*) **from** R **group by** A

to capture its precise frequency distribution. In fact, multiple such queries are usually optimized and executed together to construct multiple histograms, sharing the access to relation R. Processing the above query is very efficient if an index exists on A. Otherwise, it may become expensive, especially when the number of different values in A is high. Since statistics collection is an infrequent operation, this is usually acceptable. When not, however, sampling can be employed to obtain an approximation of the frequency distribution [PSC84].

4.2 Bucket Identification

Given a frequency distribution, the next step in constructing a histogram for it is identification of buckets. Identifying the optimal histogram among all serial ones takes exponential time in the number of buckets. On the other hand, identifying the optimal end-biased histogram takes only slightly over linear time in the number of buckets. Moreover, when constructing end-biased histograms where only high frequencies are picked for individual buckets, sampling can be used to combine data collection and bucket identification. This results in a particularly effective method with respect to both space and time. Something similar is done by DB2/MVS in order to identify the 10 highest frequencies in each attribute, which are maintained in its catalogs [Wan92]. This approach will not work when the distribution has relatively many high frequencies and few small ones, in which case low frequencies will be chosen for individual buckets, because there is no known efficient technique to identify the lowest frequencies in a distribution. Such distributions, however, are quite rare in practice (they are, in some sense, the reverse of Zipf distributions), so we believe that the above sampling-based bucket identification is rather effective.

4.3 Histogram Storage

The problem of storing a histogram (on an individual attribute) is essentially that of storing a two-column relation; one column for attribute values and one for frequencies. Unlike equi-width and equi-depth histograms, serial histograms are hard to manage because they group arbitrary attribute values into buckets. Since there is

usually no order-correlation between attribute values and their frequencies, storage of serial histograms essentially requires an index that will lead to the approximate frequency of every individual attribute value. If we generalize to histograms on combinations of attributes, then multi-dimensional indices become necessary, e.g., Grid-Files [NHS84], further increasing the complexity of histogram management.

On the other hand, end-biased histograms require very little storage and no index. One must only store the attribute values in the individual buckets and their corresponding frequencies. Since all the remaining attribute values belong in a single bucket, they do not have to be stored explicitly; only their average frequency is enough. When searching the histogram for the frequency of a value, if the value is not among those explicitly stored, it is assumed to be in the other bucket. The storage and retrieval efficiency gains compared to serial histograms are significant.

4.4 Histogram Usage

In principle, the frequency distribution of a single attribute may be viewed as a vector, that of a pair of attributes as a two-dimensional matrix, and so on. Based on that abstraction, we have shown that query result sizes, query result distributions, and all other quantities mentioned above can be obtained by multiplication of the appropriate matrices [Ioa93]. Since histograms are approximations of frequency distributions, they can also be thought of as vectors and matrices. Therefore, abstractly, estimation implies multiplying histograms in a linear algebra fashion. For general serial histograms, this essentially translates into joins, and is clearly impractical. For end-biased histograms, however, no real matrix multiplication algorithms or joins need to be invoked; more direct combinations of the simple structures used to hold the histograms are both sufficient and efficient.

5 Histogram Comparison

As shown above, the difference in construction costs between the serial and end-biased histograms is dramatic. Serial histograms are exponential in the number of buckets, whereas end-biased histograms are almost linear. Moreover, there is high complexity in the storage and usage of serial histograms. Finally, in several experiments, we have observed that most often the errors in the estimates based on end-biased histograms are not too far off from the corresponding (optimal) errors based on serial histograms. Examples of such comparisons are shown in Figures 1 and 2 [IP95]. These illustrate the relative accuracy's of serial, end-biased, and traditional histograms in estimating the result size of a query joining a relation with itself. In Figure 1, the estimation error is plotted as a function of the number of available buckets, while in Figure 2, it is plotted as a function of the skew in the join attribute (captured by the z parameter of the Zipf distribution). Clearly, the serial and end-biased histograms are significantly better than the others. Moreover, the end-biased histograms perform almost as well as the serial ones. Thus, as a compromise between optimality and practicality, we suggest that the optimal end-biased histograms should be used in real systems.

6 Histogram Weaknesses

As a last note, we would like to point out that there are some cases where serial and end-biased histograms will either provide really poor estimates or become very large and unmanageable. First, by their very nature, histograms make the following assumption:

> Uniform Frequency Assumption: All attribute values in a bucket are assumed to have the same frequency.

Clearly, this assumption will not fare well and will cause large errors if most of the attribute values occur with vastly different frequencies. Second, especially serial histograms have been presented as maintaining the precise set of attribute values associated with each bucket. As mentioned earlier, although this approach usually

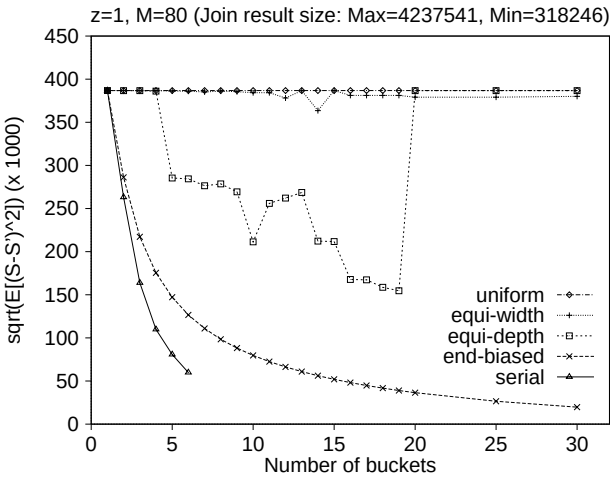

Figure 1: Error as a function of the number of buckets.

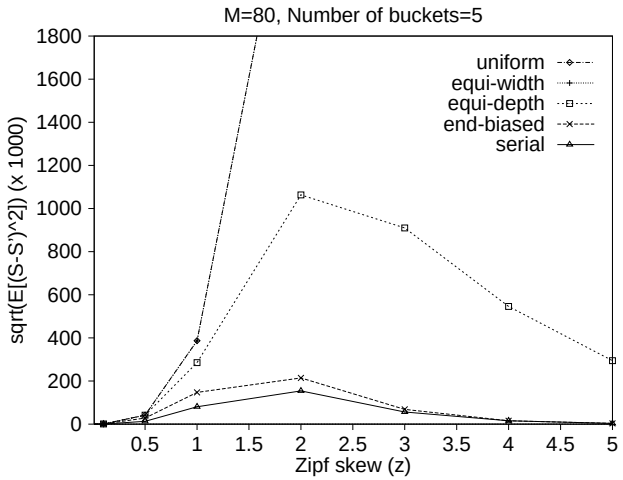

Figure 2: Error as a function of skew (z parameter of Zipf).

results in good estimates, the required storage makes it impractical. Alternatively, one could make the following assumption:

> Continuous Values Assumption: Only the boundary attribute values of each bucket are explicitly stored, and all possible intermediate values are assumed to be present in the relation.

Clearly, this assumption will generate poor results if a large number of attribute values are distributed non-uniformly in the value domain (as is often the case for floating point attributes). In both these cases, none of the histograms can approximate the frequency distribution with high accuracy, unless there are as many buckets as frequencies. We expect that such cases are not very common in real data, so histogram-based techniques will indeed be effective in estimation.

7 Summary

We have presented an overview of our work on histograms and their effectiveness in several database estimation problems. We have identified the class of serial histograms as optimal in many cases, and the class of end-biased histograms as suboptimal but far more practical. Our overall conclusions is that, even when they are not optimal, these histograms are quite effective, and can therefore be used as universal tools for many database estimation problems.

Based on the above, we intend to use histograms in building a complete query size/cost estimator. As part of the foundations of this effort, we are currently working on several problems related to histograms that are not completely solved yet. First, we are conducting an experimental study to identify robust and effective histograms for estimating result sizes of range queries. Second, we are investigating optimality of various classes of histograms for estimating access cost to secondary indices. Finally, we are looking into the effect of database updates on optimal histograms and how to automatically detect the point where histogram recalculation becomes necessary.

References

[Chr84] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM TODS*, 9(2):163–186, June 1984.

[dB78] C. de Boor. *A Practical Guide to Splines*. Springer Verlag, New York, NY, 1978.

[IC91] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the 1991 ACM-SIGMOD Conference on the Management of Data*, pages 268–277, Denver, CO, May 1991.

[IC93] Y. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM TODS*, 18(4):709–748, December 1993.

[Ioa93] Y. Ioannidis. Universality of serial histograms. In *Proc. 19th Int. VLDB Conference*, pages 256–267, Dublin, Ireland, August 1993.

[IP95] Y. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proc. of the 1995 ACM-SIGMOD Conference on the Management of Data*, pages 233–244, San Jose, CA, May 1995.

[JC85] R. Jain and I. Chlamtac. The p^2 algorithm for dynamic calculation of quantiles and histograms without sorting observations. *Communications of the ACM*, 28(10):1076–1085, October 1985.

[Koo80] R. P. Kooi. *The optimization of queries in relational databases*. PhD thesis, Case Western Reserve University, Sept 1980.

[MD88] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. of the 1988 ACM-SIGMOD Conference on the Management of Data*, pages 28–36, Chicago, IL, June 1988.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. 1984 ACM-SIGMOD Conference on the Management of Data*, pages 256–276, Boston, MA, June 1984.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Int. Symposium on Management of Data*, pages 23–34, Boston, MA, June 1979.

[Wan92] Y. Wang. Experience from a real life query optimizer. In *Proc. of the 1992 ACM-SIGMOD Conference on the Management of Data*, page 286, San Diego, CA, June 1992. Conference presentation.

The Cascades Framework for Query Optimization

Goetz Graefe

Abstract

This paper describes a new extensible query optimization framework that resolves many of the shortcomings of the EXODUS and Volcano optimizer generators. In addition to extensibility, dynamic programming, and memorization based on and extended from the EXODUS and Volcano prototypes, this new optimizer provides (i) manipulation of operator arguments using rules or functions, (ii) operators that are both logical and physical for predicates etc., (iii) schema-specific rules for materialized views, (iv) rules to insert "enforcers" or "glue operators," (v) rule-specific guidance, permitting grouping of rules, (vi) basic facilities that will later permit parallel search, partially ordered cost measures, and dynamic plans, (vii) extensive tracing support, and (viii) a clean interface and implementation making full use of the abstraction mechanisms of C++. We describe and justify our design choices for each of these issues. The optimizer system described here is operational and will serve as the foundation for new query optimizers in Tandem's NonStop SQL product and in Microsoft's SQL Server product.

1 Introduction

Following our experiences with the EXODUS Optimizer Generator [GrD87], we built a new optimizer generator as part of the Volcano project [GrM93]. The main contributions of the EXODUS work were the optimizer generator architecture based on code generation from declarative rules, logical and physical algebra's, the division of a query optimizer into modular components, and interface definitions for support functions to be provided by the database implementor (DBI), whereas the Volcano work combined improved extensibility with an efficient search engine based on dynamic programming and memorization. By using the Volcano Optimizer Generator in two applications, a object-oriented database systems [BMG93] and a scientific database system prototype [WoG93], we identified a number of flaws in its design. Overcoming these flaws is the goal of a completely new extensible optimizer developed in the Cascades project, a new project applying many of the lessons learned from the Volcano project on extensible query optimization, parallel query execution, and physical database design. Compared to the Volcano design and implementation, the new Cascades optimizer has the following advantages. In their entirety, they represent a substantial improvement over our own earlier work as well as other related work in functionality, ease-of-use, and robustness.

- Abstract interface classes defining the DBI-optimizer interface and permitting DBI-defined subclass hierarchies
- Rules as objects
- Facilities for schema- and even query-specific rules
- Simple rules requiring minimal DBI support
- Rules with substitutes consisting of a complex expression

- Rules that map an input pattern to a DBI-supplied function
- Rules to place property enforcers such as sort operations
- Operators that may be both logical and physical, e.g., predicates
- Patterns that match an entire subtree, e.g., a predicate
- Optimization tasks as data structures
- Incremental enumeration of equivalent logical expressions
- Guided or exhaustive search
- Ordering of moves by promise
- Rule-specific guidance
- Incremental improvement of estimated logical properties

The points in the list above and their effects will be discussed in this paper. While the system is operational, we have not performed any performance studies and the system is not fully tuned yet. Detailed analysis and focused improvement of the Cascades optimizer's efficiency is left for further work.

2 Optimization Algorithm and Tasks

The optimization algorithm is broken into several parts, which we call "tasks." While each task could easily be implemented as a procedure, we chose to realize tasks as objects that, among other methods, have a "perform" method defined for them. Task objects offer significantly more flexibility than procedure invocations, in particular with respect to search algorithm and search control. A task object exists for each task that has yet to be done; all such task objects are collected in a task structure. The task structure is currently realized as a last-in-first-out stack; however, other structures can easily be envisioned. In particular, task objects can be reordered very easily at any point, enabling very flexible mechanisms for heuristic guidance. Moreover, we plan on representing the task structure by a graph that captures dependencies or the topological ordering among tasks and permit efficient parallel search (using shared memory). However, in order to obtain a working system fast, the current implementation is restricted to a LIFO stack, and scheduling a task is very similar to invoking a function, with the exception that any work to be done after a sub-task completes must be scheduled as a separate task.

Figure 1 shows the tasks that make up the optimizer's search algorithm. Arrows indicate which type of task schedules (invokes) which other type; dashed arrows indicate where invocations pertain to inputs, i.e., subqueries or subplans. Brief pseudo-code for the tasks is also given in the appendix. The "optimize()" procedure first copies the original query into the internal "memo" structure and then triggers the entire optimization process with a task to optimize the class corresponding to the root node of the original query tree, which in turn triggers optimization of smaller and smaller subtrees.

A task to optimize a group or an expression represents what was called an "optimization goal" in the Volcano optimizer generator: it combines a group or expression with a cost limit and with required and excluded physical properties. Performing such a task results either in a plan or a failure. Optimizing a group means finding the best plan for any expression in the group and therefore applies rules to all expressions, whereas optimizing an expression starts with a single expression. The former is realized by invoking the latter for each expression. The latter results in transitive rule applications and therefore, if the rule set is complete, finds the best plan within the starting expression's group. The distinction between the two task types is made purely for pragmatic reasons. On the one hand, there must be a task to find the best plan for any expression in a group in order to initiate optimization

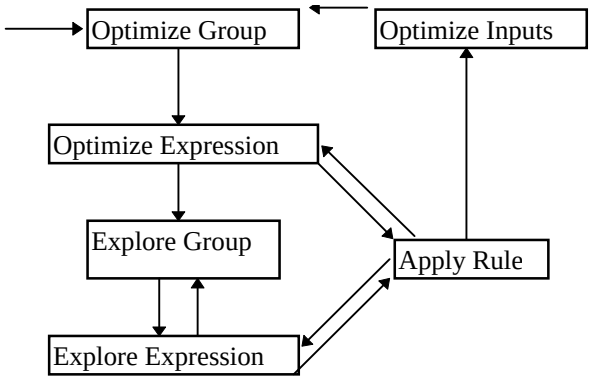

Figure 1. Optimization Tasks

of an entire query tree or a subtree after an implementation rule has been applied; on the other hand, there must be a task to optimize a single (new) expression after applying a transformation rule.

The task to optimize a group also implements dynamic programming and memorization. Before initiating optimization of all a group's expressions, it checks whether the same optimization goal has been pursued already; if so, it simply returns the plan found in the earlier search. Reusing plans derived earlier is the crucial aspect of dynamic programming and memorization. Exploring a group or an expression is an entirely new concept that has no equivalent in the Volcano optimizer generator. In the Volcano search strategy, a first phase applied all transformation rules to create all possible logical expressions for a query and all its subtrees. The second phase, which performed the actual optimization, navigated within that network of equivalence classes and expressions, applied implementation rules to obtain plans, and determined the best plan.

In the Cascades optimizer, this separation into two phases is abolished, because it is not useful to derive all logically equivalent forms of all expressions, e.g., of a predicate. A group is explored using transformation rules only on demand, and it is explored only to create all members of the group that match a given pattern. Thus, exploring a group or an expression (the distinction between these two mirrors the distinction between optimizing a group or an expression) means deriving all logical expressions that match a given pattern. The pattern, which is part of the task definition, is a subtree of the rule's antecedent or "before"-pattern.

As do optimization tasks, exploration tasks also avoid duplicate work. Before exploring a group's expressions, the task to explore a group checks whether the same pattern has already been explored for the given group. If so, the task terminates immediately without spawning other tasks. Thus, the overall effort to expand logical expressions is also reduced by dynamic programming, i.e., retaining and reusing results of earlier search effort. The decision whether or not a pattern has already been explored is made using a "pattern memory" initialized and administered by the DBI.

In order to make this discussion more concrete, consider a join associativity rule. In Volcano, all equivalence classes are completely expanded to contain all equivalent logical expressions before the actual optimization phase begins. Thus, during the optimization phase, when a join operator matches the top join operator in the rule, all join expressions for the rule's lower join are readily available so the rule can immediately applied with all possible bindings. In Cascades, these expressions are not immediately available and must be derived before the rule is applied. The exploration tasks provide this functionality; they are invoked not during a pre-optimization phase as in Volcano but on demand for a specific group and a specific pattern.

One might ask which of the Volcano technique and the Cascades technique is more efficient and more effective. The Volcano technique generates all equivalent logical expressions exhaustively in the first phase. Even if the actual optimization phase uses a greedy search algorithm, this first phase in Volcano must still be exhaustive. In the Cascades technique, this represents the worst case. If there is no guidance indicating which rule might lead

to expressions matching the given pattern, exhaustive enumeration of all equivalent logical expressions cannot be avoided. On the other hand, if there is some guidance, some of that effort can be avoided, and the Cascades search strategy seems superior. On the other hand, the same group might have to be explored multiple times for different patterns ; if so, redundant rule applications and derivations might occur. In order to avoid that, each expression in the ”memo” structure includes a bit map that indicates which transformation rules have already been applied to it and thus should not be re-applied. Thus, we believe that the Cascades search strategy is more efficient because it explores groups only for truly useful patterns. In the worst case, i.e., without any guidance, the efficiency of the Cascades search will equal that of the Volcano search strategy.

On the other hand, if such guidance is incorrect, incorrect pruning of the search space may occur and the Cascades optimizer's effectiveness might suffer. Thus, it is very important that such guidance be correct. We plan on using two techniques for guidance, which are not implemented yet. First, by inspecting the entire rule set, in particular the top operators of each rule's antecedent (”before”-pattern) and consequent (”after”-pattern, substitute), we can identify which operators can be mapped to which other operators in a single rule application. By taking the transitive closure of this reachability relationship, we can exclude some rules from consideration. Note that this transitive closure can be computed when the optimizer is generated from the rule set, i.e., only once. Second, we plan on implementing mechanisms for guidance by the DBI.

Applying a rule creates a new expression; notice that the new expression can be complex (consisting of multiple operators, as in a join associativity rule) and may be either a transformation rule (creating a new logical expression) or an implementation rule (creating a new physical expression or plan). In fact, since an operator can be both logical and physical, one rule may be both a transformation and an implementation rule. Correct rule application for such rules is guaranteed, although we expect such operators and rules to be exceptions rather than the norm.

Performing an ”apply rule” task is fairly complex. It may roughly be broken into four components. First, all bindings for the rule's pattern are derived and iterated over one by one. Second, for each binding, the rule is used to create a new expression. Note that for function rules, there may be multiple new expressions for each binding. Third, the new expressions are integrated in the ”memo” structure. Within this process, exact replicas of expressions that already exist in ”memo” are identified and removed from further consideration. Fourth, each expression that is not a duplicate of an earlier one is optimized or explored with the same goal and context that triggered the current rule application. Let us discuss these four components in turn.

Since each rule's antecedent (”before”-pattern) may be complex, the Cascades optimizer employs a complex procedure to identify all possible bindings for a rule. This procedure is recursive, with each recursive invocation for each node in the pattern. Most of its complexity serves to obtain all possible bindings for a rule's pattern. In fact, the procedure is realized as an iterator that produces the next feasible binding with each invocation. The state of this iteration is captured in the ”BINDING” class with one instance of that class for each node in the pattern. Once a binding is found, it is translated into a tree consisting of ”EXPR” nodes (note that this class is part of the DBI interface, whereas the optimizer's internal data structures are not). This copy step represents some effort, but it isolates the optimizer from the DBI methods that may be invoked for this tree. For each binding, the rule's condition function is invoked and qualifying bindings are then translated into the rule's consequent (”after”-pattern, substitute). For some rules, this is very easy and entirely left to the optimizer. For other rules, the DBI specified a function to create the substitute, and this function is invoked repeatedly to create as many substitute as possible. In other words, this function may be an iterator producing multiple substitutes in consecutive invocations. Thus, the effort of extracting a binding from the ”memo” is leveraged for multiple transformations if possible.

Each substitute expression is then integrated into the ”memo” structure. This process includes search for and detection of duplicates, i.e., expression that have been derived earlier in the optimization. This process is very similar to duplicate expression detection in both the EXODUS and Volcano optimizer generators. It is a recursive process that starts at the leaves of the substitute, which may be either query or plan tree leaves (i.e., scans) or leaf operators that denote the scope of a rewrite operation (as described as part of the DBI interface), and works upwards in the substitute towards the substitute's root; this direction is required for correct duplicate

detection. The search for duplicates is very fast as it employs a hash table using an operator and the groups of its inputs as keys.

Finally, if a substitute's root is a new expression, follow-on tasks may be initiated. If the substitute was created as part of an exploration, a task is created to explore the substitute for the same pattern. If the substitute was created as part of an optimization, the follow-on tasks depend on whether the rule was a transformation or an implementation rule, i.e., whether the substitute's root operator is a logical or a physical operator. Note, again, that an operator can be both logical and physical; thus, a rule can be both a transformation or an implementation rule. In that case, both types of follow-on tasks are created. For a logical root operator, an optimization task is created to optimize the substitute, keeping the same optimization goal. For a physical root operator, a new task is scheduled to optimize the operator's inputs and to calculated processing costs. The "optimize inputs" task is different from all other tasks. While all other tasks schedule their follow-on tasks and then vanish, this sixth task type become active multiple times. In other words, it schedules a follow-on task, waits for its completion, resumes and schedules the next follow-on task, etc. The follow-on tasks are all of the same type, which is optimizing input groups for a suitable optimization goal. Thus, like the Volcano search strategy, the Cascades search engine guarantees that only those subtrees and interesting properties are optimized that could indeed participate in a query evaluation plan. Each time after an input has been optimized, the optimize inputs task obtains the best execution cost derived, and derives a new cost limit for optimizing the next input. Thus, pruning is as tight as possible.

3 Data Abstraction and the User Interface

Developing the Cascades optimizer system required pursuing three different activities in rapid alternation. First, designing the interface between database implementor and optimizer had to focus on minimal, functional, and clean abstractions. Second, implementing a prototype optimizer as our own DBI was an exercise in exploiting the interface as effectively as possible. Third, design and implementation of an efficient search strategy was based on lessons learned during the EXODUS and Volcano projects, combined with the requirements set forth by a workshop of academic and industrial query optimization researchers and by the first user group of this software. Each of these three activities had different goals and required a different mind-set; in our internal discussions, we constantly alternated among these perspectives in order to design and develop a truly extensible and useful tool. In this section, we describe the data structuring decisions made for the interface between database implementor and the optimizer.

Users of the EXODUS and Volcano optimizer generator generators made it very clear that the interface of these system could bear improvement. Feedback from users of the Volcano optimizer generator matches our own analysis [BMG93]; therefore, we focused on (i) clean abstractions for support functions in order to enable an optimizer generator to create them from specification, (ii) rule mechanisms that permit the DBI to choose rules or functions to manipulate operator arguments (such as predicates), and (iii) more concise and complete interface specifications, both in the code and in the written documentation. Following these guidelines, we designed the following interface.

Each of the classes that make up the interface between the Cascades optimizer and the DBI is designed to become the root of a subclass hierarchy. Thus, creation of new objects of one of these classes is associated with another class. For example, creation a new "guidance" structure is associated with a "rule" object. The rule object can be of some DBI-defined subclass of the interface class "RULE," and the newly created guidance structure can be of any DBI-defined subclass of the interface class "GUIDANCE." The optimizer relies only on the method defined in this interface; the DBI is free to add additional methods when defining subclasses.

3.1 Operators and Their Arguments

Central to any database query optimizer are the sets of operators supported in the query language and in the query evaluation engine. Notice that these two sets are different; we call them logical and physical operators [Gra93]. While previous extensible operators required that these two sets be disjunct, we have abandoned this requirement. The ”class OP-ARG” in the Cascades optimizer interface includes both logical and physical operators. For each operator, one method called ”is-logical” indicates whether or not an operator is a logical operator, while a second method called ”is-physical” indicates whether or not an operator is a physical operator. In fact, it is possible that an operator is neither logical or physical; such an operator might be useful if the optimization is organized as an expansion grammar including ”non-terminals” like the Starburst optimizer [Loh88]. On the other hand, a DBI who wishes to do so can easily retain a strict separation of logical and physical operators, e.g., by defining subclasses with suitable definitions for the methods ”is-logical” and ”is-physical” and by defining all operators as subclasses of these two classes.

The definition of operators includes their arguments. Thus, no separate mechanisms are required or provided for ”argument transfer” as in EXODUS and Volcano. Notice, however, that there are two crucial facilities that permits and encourage modeling predicates etc., which had been modeled as operator arguments in all our prototypes constructed in the EXODUS and Volcano frameworks, as primary operators in the logical and physical algebra’s. First, an operator can be both logical and physical, which is natural for single-record predicates, called ”sargable” in System R [SAC79]. Second, specific predicate transformations, e.g., splitting from a complex predicate those components that can be pushed through a join, which are most easily and efficiently implemented in a DBI function rather than as rules to be interpreted by the optimizer’s search engine, can easily be realized in rules that invoke a DBI-supplied to map an expression to substitute expressions (one or more). Thus, after the EXODUS and the Volcano work has been repeatedly criticized that predicate manipulation has been very cumbersome, the Cascades optimizer offers much improved facilities.

The optimizer’s design does not include assumptions about the logical and physical algebra’s to be optimized; therefore, no query or plan operators are built into the optimizer. For use in rules, however, there are two special operators, called ”LEAF-OP” and ”TREE-OP.” The leaf operator can be used as leaf in any rule; during matching, it matches any subtree. Before a rule is applied, an expression is extracted from the search memory that matches the rule’s pattern; where the rule’s pattern has leaves, the extracted expression also has leaf operators that refer (via an array index) to equivalence classes in the search memory. The tree operator is like the leaf operator except that the extracted expression contains an entire expression, independent of its size or complexity, down to the leaf operators in the logical algebra. This operator is particularly useful in connection with function rules, which are described below.

Beyond the methods ”is-logical” and ”is-physical,” all operators must provide a method ”opt- cutoff”. Given a set of moves during an optimization task, this method determines how many of those will be pursued, obviously the most promising ones. By default, all possible moves will be pursued, because exhaustive search guarantees that the optimal plan will be found. There is also a small set of methods that must be provided only for those operators that have been declared logical. For pattern matching and for finding duplicate expressions, methods for matching and hashing are required. Methods for finding and improving logical properties are used to determine an original set of properties (e.g., the schema) and then to improve it when alternative expressions have been found (e.g., more bounds on selectivity or the output size). Finally, for exploration tasks, an operator may be called upon to initialize a pattern memory and to decide how many moves to pursue during an exploration task.

Similarly, there are some methods for physical operators. Obviously, there is a method to determine an operator’s (physical) output properties, i.e., properties of the representation. Moreover, there are three methods that compute and inspect costs. The first of these calculates the local cost of an algorithm, without any regard to the costs of its inputs. The second one combines the costs and physical properties of an algorithm’s inputs into the cost of an entire subplan. The third of these methods verifies, between optimizing two inputs of an algorithm, that the cost limit has not been exceeded yet, and computes a new cost limit to be used when optimizing the next

input. Finally, just as the last method maps an expression's cost limit to a cost limit for one of its inputs, there is a method that maps the optimization goal for an expression to an optimization goal for one of its inputs, i.e., a cost limit and required and excluded physical properties, called "input-reqd-prop." Let us discuss properties and their methods next.

3.2 Logical and Physical Properties, Costs

The interface to the interface for anticipated execution costs, the "class COST," is very simple, since instances of costs are created and returned by methods associated with other classes, e.g., operators. Beyond destruction and printing, the only method for costs is a comparison method. Similarly, the only method for the encapsulation of logical properties, the "class SYNTH-LOG- PROP," is a hash function that permits faster retrieval of duplicate expressions Since even this function does not apply to physical expressions, the encapsulation for physical properties, the "class SYNTH-PHYS-PROP," has no methods at all. The class for required physical properties, the "class REQD-PHYS-PROP," has only one method associated with it, which determines whether a synthesized physical property instance covers the required physical properties. If one set of properties is more specific than another, e.g., one indicates a result sorted on attributes "A, B, C" and the one requires sort order on "A, B" only, the comparison method returns the value "MORE." The default implementation of this method returns the value "UNDEFINED."

3.3 Expression Trees

In order to communicate expressions between the DBI and the optimizer, e.g., as queries, as plans, or in rules, another abstract data type is part of the interface, call the "class EXPR." Each instance of this class is a node in a tree, consisting of an operator and to pointers to input nodes. Obviously, the number of children in any expression node must be equal to the arity function of the node's operator. Methods on an expression node, beyond constructor, destructor, and printing, include methods to extract the operator or one of the inputs as well as a matching method, which recursively traverses two expression trees and invokes the matching method for each node's operator.

3.4 Search Guidance

In addition to pattern, cost limits, and required and excluded physical properties, rule application can also controlled by heuristics represented by instances of the "class GUIDANCE." Its purpose is to transfer optimization heuristics from one rule application to the next. Notice that costs and properties pertain to the expressions being manipulated and to the intermediate result those expressions will product when a query plan is executed; the guidance class captures knowledge about the search process and heuristics for future search activities. For example, some rules such as commutativity rules are to applied only once; for those, a simple guidance structure and a rule class are provided as part of the DBI interface, called "ONCE-GUIDANCE" and "ONCE-RULE."

Some researchers have advocated to divide a query optimizer's rule set into "modules" that can be invoked one at a time, e.g., Mitchell et al. [MDZ93]. Guidance structures can easily facilitate this design: a guidance structure indicates which module is to be chosen, and each rule checks this indication in its promise (or condition) function and then creates suitable indications when creating guidance structures for its newly created expressions and their inputs.

3.5 Pattern Memory

In addition to the search guidance, exploration effort can be restricted by use of the pattern memory. The purpose of the pattern memory is to prevent that the same group is explored unnecessarily, e.g., twice for the same pattern. There is one instance of a pattern memory associated with each group. Before a group is explored for a pattern,

the pattern memory is permitted to add the pattern to itself and is asked to determine whether or not exploration should take place. In the most simple search, in which exploration for any pattern is performed by exhaustive application of transformation rules, the pattern memory needs to contain only a Boolean, i.e., a memory whether or not the group has been explored previously. More sophisticated pattern memories would store each pattern.

Obviously, the pattern memory interacts with the exploration promise function. For the most simple promise function that always admits exhaustive search, the simple pattern memory above is suitable. It is left to the DBI to design pattern memory and promise functions most suitable to the algebra to be optimized.

Beyond checking whether a given pattern already exists in the memory, and saving it to detect a second exploration with the same pattern, the most complex method for pattern memories is to merge two pattern memories into one. This method is required when two groups of equivalent expressions are detected to be actually one, i.e., when a transformed expression already occurs in a different group in the search memory.

3.6 Rules

Next to operators, the other important class of objects in the Cascades optimizer are rules. Notice that rules are objects; thus, new ones can be created at run-time, they can be printed, etc. While other rule-based optimizers, in particular the EXODUS and Volcano optimizer generators, divide logical and physical operators as well as (logical) transformation and (physical) implementation rules into disjoint sets, the Cascades optimizer does not distinguish between those rules, other than by invoking the is-logical and is-physical methods on newly created expressions. All rules are instances of the ”class RULE,” which provides for rule name, an antecedent (the ”before” pattern), and a consequent (the substitute). Pattern and substitute are represented as expression trees, which were discussed above.

In their simplest case, rules do not contain more than that; whenever the pattern is found or can be created with exploration tasks, the substitute expression is included in the search memory. Both a rule’s pattern and substitute can be arbitrarily complex. In the EXODUS and Volcano optimizer generators, an implementation rule’s substitute could not consist of more than a single implementation operator; in the Cascades design, this restriction has been removed. The remaining restriction is that all but the substitute’s top operator must be logical operators. For example, it is possible to transform a (logical) join operator into a (physical) nested loops operator with a (logical) selection on its inner input, thus, detaching the selection predicate from the join algorithm and pushing it into the inner input tree.

For more sophisticated rules, two types of condition functions are supported. All of them consider not only the rule but also the current optimization goal, i.e., cost limit and required and excluded physical properties. First, before exploration starts, ”promise” functions informs the optimizer how useful the rule might be. There is one promise function for optimization tasks and one for exploration tasks. For unguided exhaustive search, all promise functions should return the value 1.0. A value of 0 or less will prevent the optimizer from further work for the current rule and expression. The default promise function returns 0 if a specific physical property is required, 2 if the substitute is an implementation algorithm, and 1 otherwise. If the cutoff methods associated with the operators choose exhaustive search (see above), the return value of the promise function will not change the quality of the final query evaluation plan, although it may affect the order in which plans are found, pruning effectiveness, and therefore the time an optimization takes.

Since the promise functions are invoked before exploration of subgroups, i.e., before entire expression trees corresponding to a rule’s pattern have been explored and extracted from the search memory, a ”condition” function checks whether a rule is truly applicable after exploration is complete and a complete set of operators corresponding to the pattern in the rule is available. Whereas the promise functions return a real value that expresses grades of promise, the condition function returns a Boolean to indicate whether or not the rule is applicable.

In addition to promise and condition functions, a small set of methods is associated with rules. Of course, there are constructor, destructor, and print methods, as well as method to extract the pattern, the substitute, the rule’s name, and its arity (the pattern’s number of leaf operators). The ”rule-type” method indicates whether a

rule is a simple rule (as described so far) or a function rule (to be described shortly). The ”top-match” method determines whether or not an operator in the search memory matches the top operator in the rule’s pattern; this method is the only built-in check before an promise function is invoked. The method ”opt-cases” indicates how often a physical expression is to be optimized with different physical properties. In all but a few cases, this will be one; one of the few exception is a merge-join algorithm with two equality clauses (say ”R.A == S.A and R.B == S.B”) that should be optimized for two sort orders (sorted on ”A, B” and on ”B, A”). By default, this method returns 1. The remaining methods all create new guidance structures to be used when optimizing a newly created expression and its inputs. There are two methods each for optimization and for exploration, and two each for the new expression and for its inputs, called ”opt-guidance,” ”expl-guidance,” ”input-opt-guidance,” and ”input-expl-guidance.” By default, all of them return ”NULL,” i.e., no specific guidance.

If a rule’s substitute consists of only a leaf operator, the rule is a reduction rule. If a reduction rule is applicable, two groups in the search memory will be merged. On the other hand, if a rule’s pattern consists of only a leaf operator, the rule is an expansion rule that is always applicable. The Cascades optimizer must rely on the DBI to design appropriate promise and condition functions to avoid useless transformations. Nonetheless, there is an important class of situations in which expansion rules are useful, namely the insertion of physical operators that enforce or guarantee desired physical properties. Such rules may also be called enforcer rules. Consider the inputs to a merge-join’s inputs, which must be sorted. An enforcer rule may insert a sort operation, the rule’s promise and condition functions must permit this rule only of sort order is required, and the sort operator’s ”input-reqd-prop” method must set excluded properties to avoid consideration of plans that produce their output in the desired sort order as input to the sort operator.

In some situations, it is easier to write a function that directly transforms an expression than to design and control a rule set for the same transformation. For example, dividing a complex join predicate into clauses that apply to left, right, and both inputs is a deterministic process best implemented by a single function. For those cases, the Cascades optimizer supports a second class of rules, called the ”class FUNCTION-RULE.” Once an expression is extracted that corresponds to the rule’s pattern, an iterator method is invoked repeatedly to create all substitutes for the expression. Note that the extracted expression can be arbitrarily deep and complex if the tree operator (see above) is employed in the rule’s pattern. Thus, tree operators and function rules permit the DBI to write just about any transformation. In the extreme case, a set of function rules could perform all query transformations, although that would defeat some of the Cascades framework’s purpose.

4 Future Work

Of course, there is a lot of work that should be done to make the Cascades optimizer more useful and complete. First, the optimizer has not yet gone through a thorough evaluation and tuning phase. Second, building additional optimizers based on this framework will undoubtedly show many weaknesses not yet apparent. Third, one or more generators that produce Cascades specifications from higher-level data model and algebra descriptions would be very useful. Fourth, we already know of a number of desirable improvements for the search strategy and its implementation.

The Cascades optimizer was designed to be reasonably fast, although extensibility was a more important design goal. Among the artifacts of separating the optimizer framework and the DBI’s specification of operators, cost functions, etc. are extensive use of virtual methods, a very large number of references between structures, and very frequent object allocation and deallocation. While unavoidable, there probably is room for improvement, in particular if one is willing to give up the strong separation that permits modifications of DBI code without recompiling Cascades code. Before this ”de-modularization” step is taken, however, a strong argument should be made based on a measurement study that this would indeed improve an optimizer’s performance.

5 Summary and Conclusions

Beyond a better and more robust implementation than found in the EXODUS and Volcano optimizer generators, the Cascades optimizer offers a number of advantages, without giving up modularity, extensibility, dynamic programming, and memorization explored in those earlier prototypes. First, predicates and other item operations can conveniently modeled as part of the query and plan algebra. Operators that are both logical and physical; thus, it is easy to specify operators that may appear both in the optimizer input (the query) an din its output (the plan). Function rules and the tree operator permit direct manipulation of even complex trees of item operations using DBI-supplied functions. Second, enforcers such as sorting are normal operators in all ways; in particular, they are inserted into a plan based on explicit rules. In Volcano, they were special operators that did not appear in any rule. Third, both exploration (enumeration of equivalent logical expressions) and optimization (mapping a logical to a physical expression) can be guided and controlled by the DBI. Together with the more robust implementation as required for industrial deployment, we believe that the Cascades optimizer represents a substantial improvement over earlier extensible database query optimizers.

6 Acknowledgments

The query processing group at Tandem has been very helpful in forcing me to address the hard problems unresolved in the EXODUS and Volcano optimizer generators and in finding effective and usable solutions. David Maier has been a great sounding board for ideas during the design and development of the Cascades optimizer.

7 References

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

[GrD87] G. Graefe and D. J. DeWitt, The EXODUS Optimizer Generator, Proc. ACM SIGMOD Conf., San Francisco, CA, May 1987, 160.

[Gra93] G. Graefe, Query Evaluation Techniques for Large Databases, ACM Computing Surveys 25, 2 (June 1993), 73-170.

[GrM93] G. Graefe and W. J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, Proc. IEEE Int'l. Conf. on Data Eng., Vienna, Austria, April 1993, 209.

[Loh88] G. M. Lohman, Grammar-Like Functional Rules for Representing Query Optimization Alternatives, Proc. ACM SIGMOD Conf., Chicago, IL, June 1988, 18.

[MDZ93] G. Mitchell, U. Dayal, and S. B. Zdonik, Control of an Extensible Query Optimizer: A Planning-Based Approach, Proc. Int'l. Conf. on Very Large Data Bases, Dublin, Ireland, August 1993, 517.

[SAC79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, Access Path Selection in a Relational Database Management System, Proc. ACM SIGMOD Conf., Boston, MA, May-June 1979, 23.

[WoG93] R. H. Wolniewicz and G. Graefe, Algebraic Optimization of Computations over Scientific Databases, Proc. Int'l Conf. on Very Large Data Bases, Dublin, Ireland, August 1993, 13.

A Region Based Query Optimizer Through Cascades Query Optimizer Framework

Fatma Ozcan Sena Nural Pinar Koksal Mehmet Altinel Asuman Dogac

Software Research and Development Center of TUBITAK
Dept. of Computer Engineering, Middle East Technical University
06531, Ankara Turkiye
email: asuman@srdc.metu.edu.tr

Abstract

The Cascades Query Optimizer Framework is a tool to help the database implementor (DBI) in constructing a query optimizer for a DBMS. It is data model independent and allows to code a query optimizer by providing the implementations of the subclasses of predefined interface classes. When the implementations of the required classes are provided properly, the generated optimizer produces the optimum execution plans for the queries. Although providing the complete set of rules and thus finding the optimum execution plans are beneficial for most of the queries, the query optimization time increases unacceptably for certain types of queries, e.g., for star queries. Hence it is important to be able to limit the number of alternative plans considered by the optimizer for specific types of queries by using the proper heuristics for each type. This leads to the concept of region based query optimization, where different types of queries are optimized by using different search strategies in each region.

This paper describes our experiences in developing a region based query optimizer through Cascades. Cascades' guidance structures provide the facilities required. The performance comparisons between a region based query optimizer and an exhaustive (complete rule set without heuristic guidance) query optimizer, both generated through Cascades, indicate that the region based optimizer has a superior performance. Both the sum of optimization and execution times, namely the response time, and the quality of the plans generated are investigated.

1 Introduction

The Cascades Query Optimizer Framework [Gra 94], which is being used in Microsoft's forthcoming SQL Server and Access as well as Tandem's NonStop SQL, is a tool to help the database implementor (DBI) construct a query optimizer for a DBMS. It is data model independent and allows to code a query optimizer by providing the implementations of subclasses of predefined interface classes. The interface classes include Operator, Property and Rule. When the implementations of the classes are provided properly, the optimizer generated through Cascades produces the optimum execution plan. It is an extensible system, i.e., adding or deleting rules or operators can be done easily. Cascades also provides facilities for incorporating heuristic guidance to the optimizer. These features of Cascades make it an attractive tool for developing query optimizers.

One of the earliest efforts in developing a tool for query optimization with minimal assumption on the data model is given in [Frey 87] where a rule-based description of generating equivalent query execution plans from an initial query specification is proposed. The EXODUS project of [GD 87] focuses on how to include rules into an architecture of an optimizer generator. The concepts used in EXODUS optimizer generator are, data model description as input file, rules to specify alternatives, compilation of rules into source code and separation of logical and physical operators. The drawback of the EXODUS query optimizer is the inefficiency and the ineffectiveness of its search strategy [Gra 94]. Another effort in this respect is the Volcano Query Optimizer Generator [McK 93, GM 93]. Volcano provides an efficient search engine based on dynamic programming and memorization.

However, the Volcano technique generates all equivalent logical expressions in the first phase. Even if the actual optimization phase uses a greedy search algorithm, this first phase in Volcano must still be exhaustive. In Cascades, this represents the worst case that happens when there is no heuristic guidance.

The work presented here has evolved through our experiences in developing a query optimizer for METU Object-Oriented DBMS (MOOD) [DAOD 95, Dog 95]. MOOD query optimizer [Dur 94] is developed through Volcano by using the full set of rules. Experiments with MOOD optimizer revealed the fact that the optimizer took unacceptably long for certain queries. We have identified basically two types of queries with unacceptable response times; star queries and queries with many (more than 5) selection predicates in the where clause. The poor performance for star queries is obvious; the number of alternative plans that the optimizer considers is exponential in the number of relations or classes involved. The poor performance of the queries with many selection predicates stems from the fact that there are many rules for ordering the select predicates. We have concluded that for these types of queries, instead of searching the space of alternative plans exhaustively, heuristics must be introduced. Using different sets of rules in conjunction with different heuristics for different types of queries led us to region based optimization. We have identified three types of queries to be optimized with three different strategies and implemented each strategy in a region without any interaction among the regions. This is an initial step towards the region based optimization described in [Mit 93]. We then developed such a region based optimizer through Volcano [Kok 95]. However, since it is not possible to add new control strategies to the Volcano search engine, we had to introduce outside control over Volcano, which reduced the effectiveness of the approach.

In this paper, we describe our experiences in developing the region based query optimizer through Cascades. The rule set used [BMG 93] is provided in the Appendix. These rules are sufficient to optimize both relational and object-oriented queries. The performance comparisons between a region based query optimizer and an exhaustive query optimizer, both generated through Cascades, indicate that the region based optimizer has a superior performance.

The paper is organized as follows: Section 2 contains a brief summary of the previous literature that is directly related to our work. A short summary of Cascades Query Optimizer Framework is given in Section 3. In Section 4, generating a region based optimizer through Cascades is described. Section 5 presents the performance comparisons between a region based query optimizer and an exhaustive query optimizer, both generated through Cascades. Finally, Section 6 contains the conclusions.

2 Related Work

We begin by noting that some instances of the query optimization problem are NP-complete [IK 84]. In [OL 90], it is shown that the complexity of optimizing the order of join operations is dependent upon the shape of the query. The shape of the query indicates how tables are connected with predicates. In linear queries, tables are connected by binary predicates in a straight line; whereas in star queries, a table at the center is connected by binary predicates to each of the other surrounding tables.

The computational complexity (i.e. the number of joins that must be considered when using dynamic programming for optimization) of linear queries with composite inners (bushy trees) is (N^3-N)/6. If the composite inners are not considered the complexity reduces to $(N-1)^2$. On the other hand, using dynamic programming to optimize a star query with N quantifiers requires evaluating $(N-1)2^{N-2}$ feasible joins [OL 90]. This study indicates that it is not feasible to consider all possible alternative plans for star queries. Yet, certain other types of queries might benefit a lot from considering all possible plans. The space of alternative plans must therefore be adjustable for each type of query. This leads to the concept of region based query optimization.

In [Mit 93, MZD 92, MDZ 93, MDZ 94] an architecture for region based extensible processing and optimization of queries is proposed. An Epoq optimizer is a collection of concurrently available region modules, each of which embodies one strategy for the optimization of query expressions. Different regions often accomplish dif-

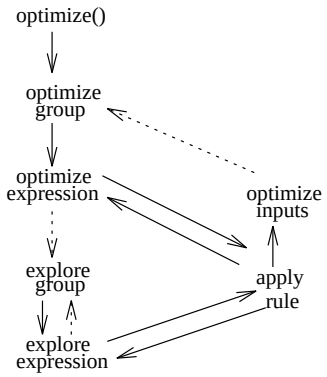

Figure 1: Cascades task structure

ferent query transformation tasks, but regions may also represent different strategies for accomplishing the same task in different ways. The Epoq architecture integrates the regions through a common interface for the region modules, and a global control that combines the actions of subordinate regions to process a given query. The region modules are organized hierarchically, with a parent region controlling its subordinate regions as though they were a collection of transformations. The regions define, through their interface, the characteristics of queries they can process, goals for the transformation of queries, and the characteristics of result queries. A higher level control uses this information to plan a sequence of region executions to process a given query expression.

In [Loh 88, HFLP 89], in the framework of the extensible query processing in Starburst, productions of a grammar are used to define query execution plan alternatives. The terminals of this grammar are base-level database operations on tables and the nonterminals are defined declaratively by production rules which combine those operations into execution plans. Each of these productions produce a set of alternative plans, each having a vector of properties, including the estimated cost. In addition, producti ons can require certain properties of their inputs, such as sortedness.

3 Cascades Query Optimizer Framework

In Cascades [Gra 94], the optimization algorithm is broken into several parts, which are called tasks. A task is realized as an object and a task object exists for each task that has yet to be done. All the tasks are collected in a task structure, a last-in-first-out stack. In Figure 1, tasks that make up the optimizer's search engine are shown. The "optimize" procedure first copies the original query into an internal structure, namely the "memo" structure, which holds all the equivalent logical and physical expressions. Then "optimize" triggers the entire optimization process with a task, namely "opt_group", to optimize the class corresponding to the root of the original query tree, which in turn triggers optimization of smaller subtrees. A task to optimize a group, which is a collection of equivalent expressions, or a single expression combines a group or an expression with a cost limit and with required and excluded physical properties. Performing such a task results in either finding the best plan, or failure. Exploring a group or an expression is a new concept that has no equivalent in Volcano. In Volcano all rules are applied in the first phase to create all possible logical expressions of a query and its subtrees. In the second phase, implementation rules are applied to these logical expressions to obtain plans and the best plan is chosen using branch and bound pruning. In Cascades, this separation into two phases is abolished, since it is not useful to derive all logically equivalent expressions. A group is explored by applying rules only on demand.

While the EXODUS and Volcano optimizer generators had the concept of support functions, Cascades is completely based on C++ subclasses. Using the object-oriented paradigm, Cascades provides a clear interface between the optimizer and the DBI supplied functions. Each of the classes that make up the interface between

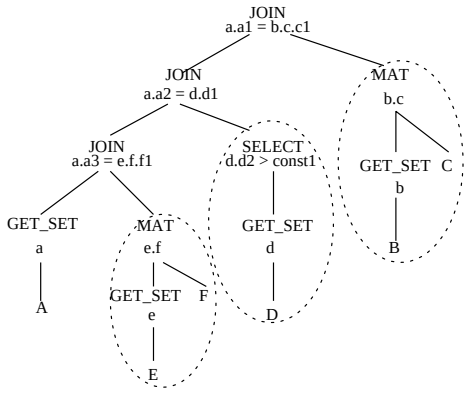

Figure 2: Example star query

the optimizer and DBI is designed to be the root of a subclass hierarchy. The DBI creates a new optimizer by providing the implementations of these subclasses.

In Cascades, an operator can be both logical and physical. For each operator, one method indicates whether an operator is a logical operator, while a second method determines if an operator is a physical operator. The definition of operators includes their arguments, thus, no separate mechanism is provided for ”argument transfer”. All operators must provide a method which determines how many of the given set of rules will be applied. Methods for matching, finding and improving logical properties must be provided by the DBI for operators that are declared to be logical. Similarly, for physical operators, methods for finding an operator’s output properties and for computing and inspecting an operator’s cost must be supplied by the DBI. Another method that maps an expression’s cost limit to a cost limit for one of its inputs is also required.

Another important class of objects in Cascades Optimizer Framework is the class RULE. Cascades does not distinguish between (logical) transformation and (physical) implementation rules. All rules have a name, an antecedent (the pattern) and a consequent (the substitute) both of which can be arbitrarily complex. Two types of condition functions are supported which consider the rule and the current optimization goal, i.e., the cost limit and the required and excluded properties. In addition to cost limits, required and excluded properties, rule application can also be controlled by heuristics. For this purpose, before optimization starts, ”promise” functions inform the optimizer how useful the rule might be. When there is no guidance, all promise functions should return 1 to indicate that the rule will be pursued. A value 0 or less will prevent the application of the rule. All promise and condition functions must be supplied by the DBI. Guidance information is passed to the functions by the help of the guidance class. The guidance class captures knowledge about the search process and the heuristics for future search activities and it is handled by the DBI.

4 A Region Based Query Optimizer Generated Through Cascades

As described in Section 1, we have identified three query optimization regions. In order to classify the queries into these regions, the following criterion is used: A query that has two or more join operators which have the same bind variable as one of their operands is classified as a star query. If a query is not in this region and has five or more select operators then it is classified as a select query. Otherwise, it is classified as a default query and optimized with the complete rule set. Currently a query falls only in one region and is optimized in that region without any interaction with the other regions. In the following, the heuristics introduced for star and select regions along with their implementation in Cascades are presented.

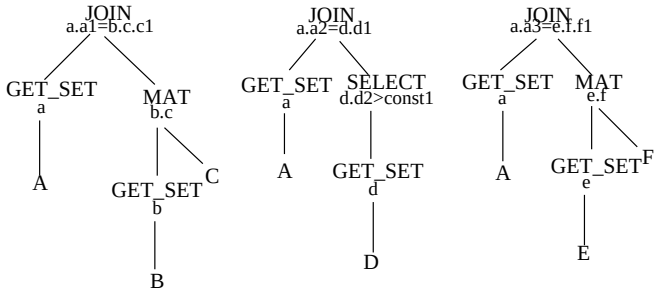

Figure 3: Subqueries of example star query

4.1 Star Region

It has been observed that with exhaustive search strategy Cascades Optimizer Framework spends too much time to optimize star queries, and most of this time is spent in ordering the join classes that cause the query to be star shaped. The ordering of join classes is provided by commutativity and associativity rules of join classes (Rules 8,9). If these two rules are disabled, optimization time can be saved. However, the plans generated by disabling these rules result in large deviations from the optimum plan therefore heuristics must be used. The heuristic that we have introduced is as follows: we assume the linear subpart of a star query as a unit of processing and we process a star query by executing the linear subqueries in the order from the least costly to the most costly. The intuition behind this heuristic is the following: executing a linear subquery will reduce the size of a partial result and this will be achieved in the least expensive way by executing the least expensive subqueries first. Yet, not only the size of a partial result of a join, but join selectivity also affects the order of joins. We have developed another heuristic to consider join selectivity and join cost together which is not described in this paper due to space limitations, [ODE 95].

In star region, each linear subquery of a given query, (shown with dotted lines in Figure 2) is thought as a unit of processing and optimization of this linear part with associativity and commutativity of join rules enabled (Rules 8,9), takes polynomial time. We need to optimize these linear subqueries (Figure 3) one at a time according to our heuristic. However, calling the ”optimize” routine once for each of these subqueries is not efficient since a new ”memo” structure is created for each subquery, making it impossible to share common subqueries and to use previously generated physical plans. Therefore, the search engine of Cascades is extended to allow the subqueries to be optimized one at a time without losing the previously generated physical plans. In order to achieve this, the task structure of Cascades is exploited. In Cascades, the task ”opt_group” takes a group to optimize, combines the group with a cost limit, required and excluded physical properties and returns either a plan or failure. In its original form, optimization begins by triggering an ”opt_group” task for the class corresponding to the root of the input query. We have modified this concept and for star queries we triggered an ”opt_group” task for each subquery. After the optimization of subqueries is completed, the subquery with the minimum estimated cost is chosen and joined with the other subqueries, as shown in Figure 4. Then in the next pass, these newly created subqueries are optimized. While optimizing a group corresponding to one of these newly created subqueries, previously generated physical plans are used, thus, no rule application take place in these parts, since Cascades explores a group only on demand, i.e. when there is no plan satisfying the current optimization goal. This process continues until no subquery is left. Thus, there are as many passes as the initial number of subqueries.

The modified form of the ”optimize” routine of Cascades is given in Algorithm 1.

Algorithm 1: Modified ”optimize” routine of Cascades

```
optimize(qry,guidance){
    if guidance.region is star_region{
      create_subqueries(qry) //creates subqueries in the ”memo” structure
```

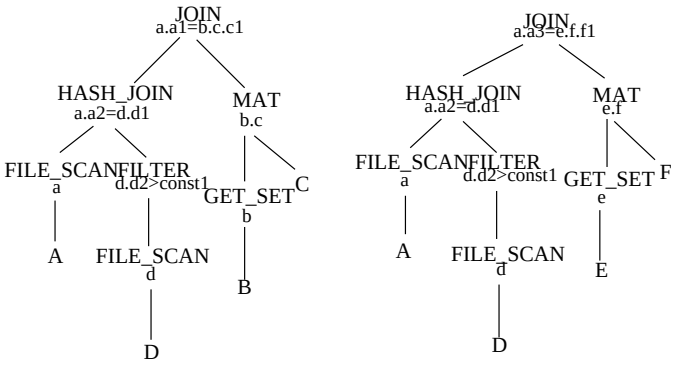

Figure 4: Subqueries after the first pass

```
    set pass_count to number of subqueries
  }
  for i= 1 to pass_count do{
    if guidance.region is star_region{
      for j=1 to current number of subqueries
          push "opt_group" task for jth subquery to task_list
    }
    else //not star region
      push "opt_group" task for the root of the query
    while task_list is not empty
      perform task
    if guidance.region is star_region{
      find the subquery with the minimum cost
      modify_memo() //update "memo" structure such that the subquery
                        with the minimum cost is joined with other subqueries
    }
  }
  return plan
}
```

4.2 Select Region

In the complete rule set given in the Appendix [BMG 93], there are many rules for select classes (Rules 1,2,3,4,5,6,7,10,15,16). These transformations increase the optimization time a lot for queries with many select predicates. We have further noted that some of these rules can be disabled by using effective heuristics instead. In the following, we describe these heuristics through examples. To be able to apply these heuristics, certain modifications are required in the input query tree. The first modification is placing the select operators at the top of the tree. This eliminates the need for the rules (Rules 4,10) that move selections up in the tree. Second modification requires the decomposition of select predicates beforehand so that there is no need for the rule (Rule 1) that decomposes them. Another rule that can be disabled is Rule 3 for the commutativity of select operators. Yet, when all these rules are disabled, there will be large deviations from the optimal plan. As an example consider the query tree of Figure 5.a and assume the corresponding optimum execution tree is as given in Figure 5.b. When all these rules are disabled, the optimizer can not generate the tree of Figure 5.b because SELECT_1 cannot be moved beneath SELECT_2. This problem can be eliminated by introducing the following heuristics: First sort the select operators for the same range variable according to their number of path expression elements. The intuition behind this is, the path with fewer expression elements can be moved beneath in the query tree more deeply during the

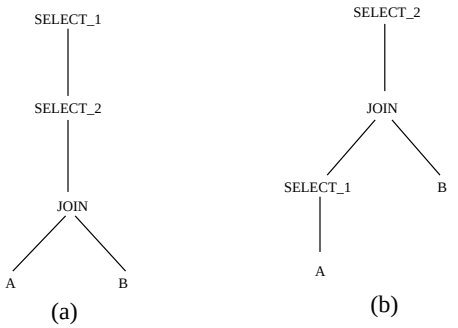

Figure 5: Select commutativity example

optimization. Next, the select operators are sorted according to their selectivities by preserving the relative order imposed by the first sorting. The intuition behind this heuristic is to let more selective predicate go down the query tree. With these sortings, it is possible for Cascades to generate the optimum execution plan for the Figure 5 example even when the rules mentioned above are disabled.

After the required modifications are performed on the query tree (i.e. moving the select predicates and decomposing), the query is optimized in the select region. Two pass optimization is performed by calling the "optimize" routine twice. In the first pass, all select rules (Rules 1,2,3,4,5,6,7,10) are disabled and the join and materialize operators are ordered. At the end of this pass, the "optimize" routine of Cascades returns a physical plan. This physical plan is converted to a logical query which the "optimize" routine takes as input, and is optimized again in the second pass. Although it is possible to make two pass optimization inside the "optimize" routine, it is not very efficient since in the first pass, Cascades creates a lot of equivalent logical and physical expressions, which are not required in the next pass, and if these are not deleted, a large number of rules will match to these large number of expressions. Therefore, the "optimize" routine is called twice making it possible to use a new "memo" structure in the second pass in which the select operators are ordered by disabling join and mat rules (Rules 8,9,11,12). The overall approach is presented in Algorithm 2.

Algorithm 2: Region based optimizer

```
Region_based_optimizer ( QUERY qry, PLAN plan){
    create_operator_table(qry,op_table) //creates a table to count select and join classes
    decide_qry_type(op_table) // decides the query type
    if query is star_query
      set guidance.region to star_region
    else if query is select_query
      set guidance.region to select_region
    else
      set guidance.region to default_region
    if guidance.region is select_region{
      order_query(qry) //reorders the query such that select operators are at the top of the query
      set guidance.count to 0 //to order joins, all select rules are disabled
      order_selects (qry) // sort selects
      plan = optimize (qry,guidance) // a plan is found
      convert_qry (plan,qry) //converts the coming physical plan into an equivalent logical query
      set guidance.count to 1 //enables select rules while disabling join and mat rules
    }
```

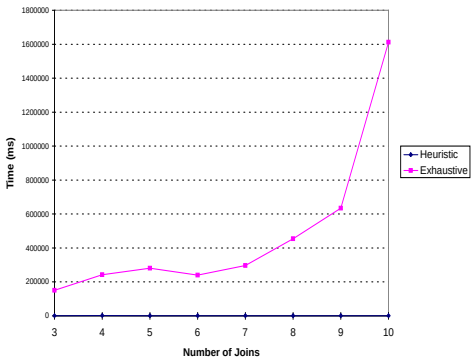

Figure 6: Optimization times for the star region

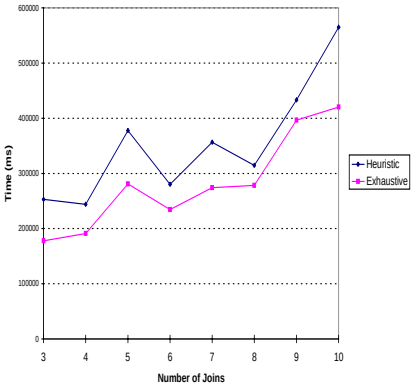

Figure 7: Estimated execution times for the star region

plan = optimize (qry,guidance)

}

5 Performance Evaluation

This section presents the performance comparison of the region based and an exhaustive optimizer, both generated through Cascades. Their optimization and response times and the quality of the plans generated are compared. In the experiments, a workstation running Microsoft NT with 64 MB main memory and 250 MB swap space is used. The queries are generated by a random query generator, and both optimizers are run with the same set of queries. The optimization times are obtained from the system and the execution times are estimated using the same cost functions for both optimizers. The results are presented below.

5.1 Star Region

For the experiments in the star region, 80 star queries are generated randomly. Number of join operators in this query set, range from 3 to 10, and for each query type 10 queries are generated, and the average of these 10 values are used. Figure 6, Figure 7 and Figure 8 depict the results of the experiments. As shown in Figure 6, optimization time difference between two optimizers increases dramatically as the number of join operators increases. Although the plans generated using the region based optimizer are slightly worse than the optimal (Figure 7), the total response time is better in the region based optimizer (Figure 8). Since the aim of query optimization is to minimize the response time, these results justify the heuristic used in the star region of the region based optimizer.

5.2 Select Region

For the experiments on queries with many select operators, 50 select queries are generated randomly with the number of select predicates ranging from 5 to 9 and with 2 join and 3 materialize operators. Comparison of the optimization time, quality of the plans generated, and response times are shown in Figure 9, Figure 10 and Figure 11 respectively. The region based optimizer spends less time to optimize queries than the exhaustive optimizer (Figure 9). The quality of the plans generated using the region based query optimizer is slightly worse than that of the optimal ones as shown in Figure 10. This is an expected result, since when some rules are disabled, the

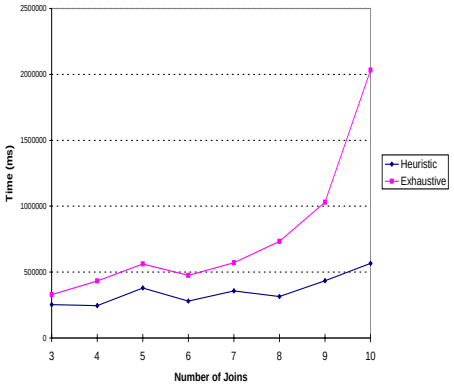

Figure 8: Response times for the star region

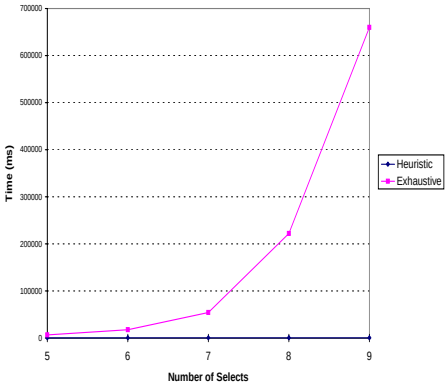

Figure 9: Optimization times for the select region

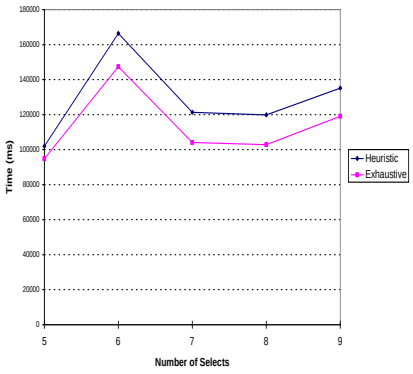

Figure 10:Estimated execution times for the select region

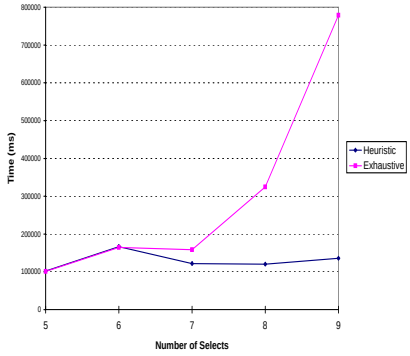

Figure 11: Response times for the select region

generated plans deviate from the optimal. However, the total response times of queries are better for the region based optimizer (Figure 11). These results also indicate that the heuristic suggested is an effective one.

6 Conclusions

A region based optimizer with three optimization regions, each of which optimizes different queries with different strategies, is generated through Cascades Query Optimizer Framework. Heuristic guidance feature of Cascades proved to be very useful in this. The performance of the region based optimizer is compared with one that uses the complete rule set, also generated through Cascades. Our experiments show that although with the region based optimizer the quality of the plans are not as good as the plans generated by the exhaustive optimizer, optimization time gain is substantial. For this reason, region based optimizer's total time for queries, i.e., the query response times are better.

References

[BMG 93] Blakeley, J. A., McKenna, W. J., Graefe, G., "Experiences Building the Open OODB Query Optimizer", Proc. of the ACM SIGMOD Conf., 1993.

[DAOD 95] Dogac, A., Altinel, M., Ozkan, C., Durusoy, I., "Implementation Aspects of an Object-Oriented DBMS", in ACM SIGMOD Record, Vol.24, No.1, March 1995.

[Dog 95] Dogac, A., Altinel, M., Ozkan, C., Durusoy, I., Altintas, I., "METU Object-Oriented DBMS Kernel", 6th International Conference on Database and Expert Systems Applications, London, September 1995 (Lecture Notes in Computer Science, Springer Verlag 1995).

[Dur 94] Durusoy, I., "MOOD Query Optimizer", M.Sc. Thesis, Dept. of Computer Eng., Middle East Technical Univ., Ankara, Turkey, 1994.

[Frey 87] Freytag, J.C., "A Rule-based View of Query Optimization", in Pr oc. of ACM SIGMOD Conf., 1987.

[GD 87] Graefe, G., DeWitt, D. J., "The EXODUS Optimizer Generator", in Proc. of ACM SIGMOD Conf., 1987.

[GM 93] Graefe, G., McKenna, J. W., "The Volcano Optimizer Generator:Extensibility and Efficient Search", Proc. IEEE Conf. on Data Eng., Vienna Austria,1993.

[Gra 94] Graefe, G.,"Query Optimization in the Cascades Project", unpublished manuscript, 1994.

[HFLP 89] Haas, L.M.,Freytag, J.C.,Lohman, G.M.,Pinaresh, H., "Extensible Query Processing in Starburst", in proceedings of ACM SIGMOD Conf., 1989

[IK 84] Iberaki, T., and Kameda, T., "On the Optimal Nesting Order for Computing N-Relational Joins", ACM Transitions on Database Systems, Vol. 9, No. 3, 1984.

[Kok 95] Koksal, P., "Design and Implementation of a Region Based Query Optimizer for Object-Oriented DBMSs", M.Sc. Thesis, Dept. of Computer Eng., Middle East Tech. Univ., Ankara, Turkey, 1995.

[Loh 88] Lohman, G.M., "Grammar-like Functional Rules for Representing Query Optimization Alternatives", in proceedings of ACM SIGMOD Conf., 1988

[McK 93] McKenna, W.J., "Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator", Ph. D. Thesis, Univ. of Colorado, 1993.

[MDZ 93] Mitchell, G., Dayal, U., and Zdonik, B.S., "Control of an Extensible Query Optimizer: A Planning-Based Approach", Proc. of Intl. Conf. on Very Large Databases, 1993.

[MDZ 94] Mitchell, G., Dayal, U., and Zdonik, B.S., "Optimization of Object-Oriented Queries: Problems and Approaches", in Advances in Object-Oriented Database Systems, Springer Verlag, 1994.

[Mit 93] Mitchell, G., "Extensible Query Processing in an Object-Oriented Database" PhD thesis, Brown University, 1993.

[MZD 92] Mitchell, G., Zdonik, S., and Dayal, U., "An Architecture for Query Processing in Persistent Object Stores", Proc. of the Hawaii Intl. Conf. on System Sciences, 1992.

[ODE 95] Ozkan, C., Dogac, A., Evrendilek, C., "A Heuristic Approach for Optimization of Path Expressions in Object-Oriented Query Languages", 6th International Conference on Database and Expert Systems Applications, London, September 1995 (Lecture Notes in Computer Science, Springer Verlag 1995).

[OL 90] Ono, K., Lohman, G. M., "Measuring the Complexity of Join Enumeration in Query Optimization", Proc. of Intl. Conf. on Very Large Databases, 1990.

Appendix: The Complete Rule Set

1. Select (op,pred) –> Select (Select (op,pred1) , pred2)
2. Select (Select(op,pred1),pred2) –> Select (op,pred)
3. Select (Select(op,pred1),pred2) –> Select (Select (op,pred2),pred1)
4. Mat (Select (op,pred)) –> Select (Mat (op),pred)
5. Select (Mat (op), pred2) –> Mat (Select (op,pred))
6. Select (Join (op1,op2,pred1),pred2) –> Join (Select (op1,pred2),op2,pred1)
7. Select (Join (op1,op2,pred1),pred2) –> Join (op1,Select(op2,pred2),pred1)
8. Join (op1,op2,pred) –> Join (op2,op1,pred)
9. Join (Join (op1,op2,pred1),op3,pred2) –> Join (op1,Join (op2,op3,pred3),pred4)
10. Join (Select (op1,pred1) ,op2,pred2) –> Select (Join (op1,op2,pred2),pred1)
11. Mat1 (Mat2 (op)) –> Mat2 (Mat1 (op))
12. Mat (op) –> Join (Get_set,op1,pred)
13. Get_set –> File_scan
14. Mat (op) –> Traverse (op)
15. Select (op,pred) –> Filter (op,pred)
16. Select (op,pred) –> B_tree_select (op,pred)
17. Join (op1,op2,pred) –> Merge_join (op1,op2,pred)
18. Join (op1,op2,pred) –> Hash_join (op1,op2,pred)
19. Join (op1,op2,pred) –> Nested_loop_join (op1,op2,pred)
20. Join (op,Get_set,pred) –> Ptr_hh_join (op1,pred1)

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 transformation 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 also important 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).

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.

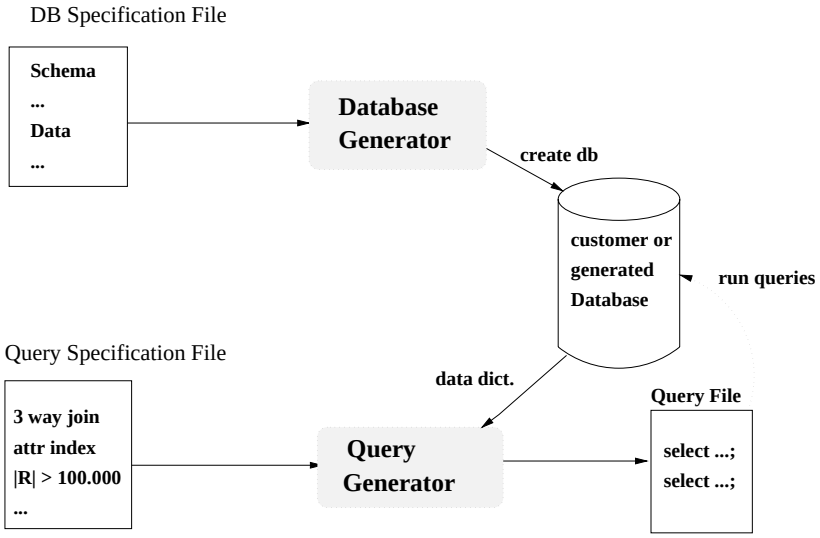

Figure 1: Overview

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).

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 intermediate 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 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).

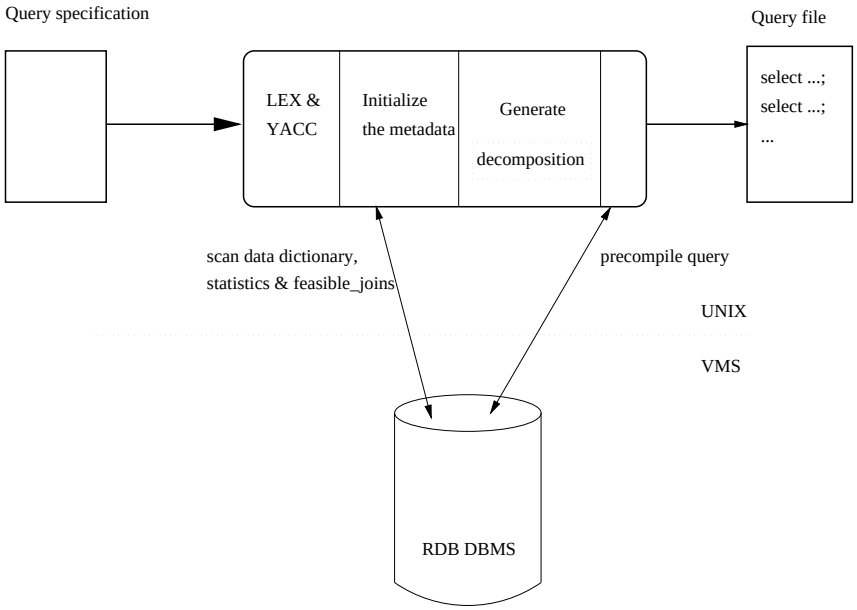

Figure 2: the query generator

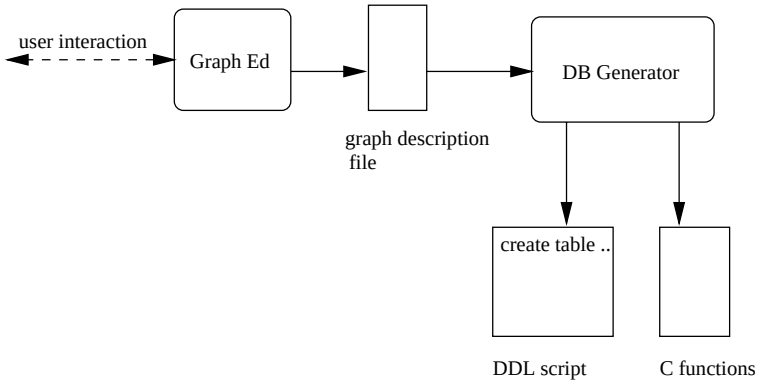

Figure 3: Database Generator

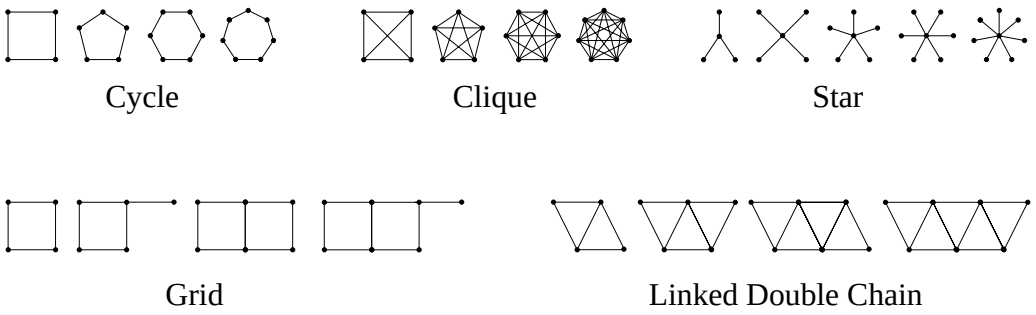

Figure 4: graph classes

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.

CALL FOR PAPERS

DATA ENGINEERING

12th International Conference on Data Engineering

February 26 - March 1, 1996
New Orleans, Louisiana, USA
Sponsored by the IEEE Computer Society

IEEE

SCOPE

Data Engineering deals with the use of engineering disciplines in the design, development and evaluation of information systems for different computing platforms and system architectures. The 12th Data Engineering Conference will provide a forum for the sharing of original research results and practical engineering experiences among researchers and practitioners interested in all aspects of data and knowledge management. The purpose of the conference is to share solutions to problems that face our information-oriented society and to identify new challenges and directions for future research.

TOPICS OF INTEREST

The topics of interest include but are not limited to:

- Data Engineering and Information Superhighways
- Mobile Computing
- Network Databases and Security
- Knowledge Mining and Discovery
- Interoperability of Heterogeneous Information Systems
- Distributed Transaction Management
- Query processing and Optimization
- Virtual Enterprise Management
- Work Management and Simulation
- Multimedia Systems and Applications
- Open Architecture and Extensible Systems
- Visualization and User Interface
- Active Database/Knowledge Base Systems
- Intelligent and Deductive Systems
- Parallel and Distributed Database Systems
- Object-oriented Databases and Software Engineering
- Real-time, Temporal and Spatial Systems
- Industrial Challenges in Data Engineering
- Scientific and Statistical Databases
- Information Systems for Engineering Application
- Databases and Information Retrieval

PAPER SUBMISSION

Six copies of original papers not exceeding 6000 words (25 double spaced pages) should be submitted by May 31, 1995 to program chair:

Stanley Y. W. Su
Database Systems R&D Center
470 CSE Building
University of Florida
P. O. Box 116125
Gainesville, FL 32611-6125
E-mail: icde96@cis.ufl.edu
Tel. (904) 392-2680, FAX: (904) 392-1220

Authors should explicitly specify in the cover page one or two topics of interest most relevant to the submission. Submissions for the industrial program should also be specified in the cover page.

ORGANIZING COMMITTEE

General Chair: Marek Rusinkiewicz, University of Houston
Program Chair: Stanley Y. W. Su, University of Florida
Industrial Program Chair: Umesh Dayal, Hewlett-Packard Laboratories
Panel Program Chair: Arie Segev, University of California at Berkeley
Tutorial Program Chair: Witold Litwin, University of Paris

PROGRAM VICE CHAIRS

Data Engineering and Information Super Highways
Gunter Schlageter, Univ. of Hagen

Extensible and Active Database/Knowledge Base Systems
Sharma Chakravarthy, Univ. of Florida

Interoperability of Heterogeneous Information Systems
Dennis McLeod, Univ. of Southern California

Parallel and Distributed Database Systems
Chaitanya K. Baru, IBM Toronto Labs

Scientific and Statistical Databases and Applications
Arie Shoshani, Lawrence Berkeley Laboratory

Object-oriented Databases and Software Engineering
Luqi, Naval Postgraduate School

Multimedia Systems and Applications
William Grosky, Wayne State Univ.

Real-time, Temporal and/or Spatial Systems
Alex Buchmann, Technical Univ. Darmstadt

INDUSTRIAL, PANEL &TUTORIAL PROGRAMS

There will be a series of sessions focused on issues relevant to practitioners of information technology. Proposals for the industrial program, panels and the tutorial program are being sought. Proposers should submit a short write-up to the persons in charge of the Industrial, Panel and Tutorial programs.

PUBLICATIONS & AWARDS

All accepted papers will appear in the Proceedings published by IEEE Computer Society. The authors of selected papers will be invited to submit extended versions for possible publication in the *IEEE Transactions on Data and Knowledge Engineering* and in the *Journal of Distributed and Parallel Databases*. An award will be given to the best paper. A separate award honoring K.S. Fu will be given to the best student paper (authored solely by students).

IMPORTANT DATES

- Paper, Panel, and Tutorial submissions: May 31, 1995
- Notification of acceptance: October 2, 1995
- Tutorials: February 26-27, 1996
- Conference: February 28 - March 1, 1996

EUROPEAN COORDINATOR

- Keith G. Jeffery, Rutherford Appleton Lab

FAR EAST COORDINATOR

- Ron Sacks-Davis, CITRI

EXHIBITS PROGRAM

- Wei Sun, Florida International Univ.

PUBLICITY CHAIR

- Dimitrios Georgakopoulos, GTE Laboratories

FINANCIAL CHAIR

- Stephen Huang, Univ. of Houston

REGISTRATION

- to be named

LOCAL ARRANGEMENTS

- William Buckles, Tulane Univ.

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398