An Overview of Query Optimization in Relational Systems

Surajit Chaudhuri
Microsoft Research
One Microsoft Way
Redmond, WA 98052
+1-(425)-703-1938
surajitc@microsoft.com

1. OBJECTIVE

There has been extensive work in query optimization since the early '70s. It is hard to capture the breadth and depth of this large body of work in a short article. Therefore, I have decided to focus primarily on the optimization of SQL queries in relational database systems and present my biased and incomplete view of this field. The goal of this article is not to be comprehensive, but rather to explain the foundations and present samplings of significant work in this area. I would like to apologize to the many contributors in this area whose work I have failed to explicitly acknowledge due to oversight or lack of space. I take the liberty of trading technical precision for ease of presentation.

2. INTRODUCTION

Relational query languages provide a high-level "declarative" interface to access data stored in relational databases. Over time, SQL [41] has emerged as the standard for relational query languages. Two key components of the query evaluation component of a SQL database system are the *query optimizer* and the *query execution engine*.

The query execution engine implements a set of *physical operators*. An operator takes as input one or more data streams and produces an output data stream. Examples of physical operators are (external) sort, sequential scan, index scan, nested-loop join, and sort-merge join. I refer to such operators as physical operators since they are not necessarily tied one-to-one with relational operators. The simplest way to think of physical operators is as pieces of code that are used as building blocks to make possible the execution of SQL queries. An abstract representation of such an execution is a *physical operator tree*, as illustrated in Figure 1. The edges in an operator tree represent the data flow among the physical operators. We use the terms physical operator tree and *execution plan* (or, simply *plan*) interchangeably. The execution engine is responsible for the execution of the plan that results in generating answers to the query. Therefore, the capabilities of the query execution engine determine the structure of the operator trees that are feasible. We refer the reader to [20] for an overview of query evaluation techniques.

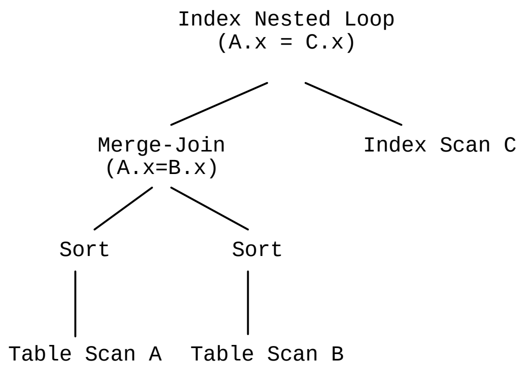

Figure 1. Operator Tree

The query optimizer is responsible for generating the input for the execution engine. It takes a parsed representation of a SQL query as input and is responsible for generating an *efficient* execution plan for the given SQL query from the space of possible execution plans. The task of an optimizer is nontrivial since for a given SQL query, there can be a large number of possible operator trees:

- The algebraic representation of the given query can be transformed into many other logically equivalent algebraic representations: e.g.,
  `Join(Join(A,B),C)= Join(Join(B,C),A)`
- For a given algebraic representation, there may be many operator trees that implement the algebraic expression, e.g., typically there are several join algorithms supported in a database system.

Furthermore, the throughput or the response times for the execution of these plans may be widely different. Therefore, a judicious choice of an execution by the optimizer is of critical importance. Thus, query optimization can be viewed as a difficult search problem. In order to solve this problem, we need to provide:

- A space of plans (*search space*).
- A *cost estimation* technique so that a cost may be assigned to each plan in the search space. Intuitively, this is an estimation of the resources needed for the execution of the plan.
- An *enumeration algorithm* that can search through the execution space.

A desirable optimizer is one where (1) the search space includes plans that have *low cost* (2) the costing technique is *accurate* (3) the enumeration algorithm is *efficient*. Each of these three tasks is nontrivial and that is why building a good optimizer is an enormous undertaking.

We begin by discussing the System-R optimization framework since this was a remarkably elegant approach that helped fuel much of the subsequent work in optimization. In Section 4, we will discuss the search space that is considered by optimizers. This section will provide the forum for presentation of important algebraic transformations that are incorporated in the search space. In Section 5, we address the problem of cost estimation. In Section 6, we take up the topic of enumerating the search space. This completes the discussion of the basic optimization framework. In Section 7, we discuss some of the recent developments in query optimization.

3. AN EXAMPLE: SYSTEM-R OPTIMIZER

The System-R project significantly advanced the state of query optimization of relational systems. The ideas in [55] have been incorporated in many commercial optimizers continue to be remarkably relevant. I will present a subset of those important ideas here in the context of Select-Project-Join (SPJ) queries. The class of SPJ queries is closely related to and encapsulates *conjunctive queries*, which are widely studied in Database Theory.

The search space for the System-R optimizer in the context of a SPJ query consists of operator trees that correspond to *linear* sequence of join operations, e.g., the sequence `Join(Join(Join(A,B),C),D)` is illustrated in Figure 2(a). Such sequences are logically equivalent because of associative and commutative properties of joins. A join operator can use either the nested loop or sort-merge implementation. Each scan node can use either index scan (using a clustered or non-clustered index) or sequential scan. Finally, predicates are evaluated as early as possible.

The cost model assigns an estimated cost to any partial or complete plan in the search space. It also determines the estimated size of the data stream for output of every operator in the plan. It relies on:

(a) A set of statistics maintained on relations and indexes, e.g., number of data pages in a relation, number of pages in an index, number of distinct values in a column

(b) Formulas to estimate selectivity of predicates and to project the size of the output data stream for every operator node. For example, the size of the output of a join is estimated by taking the product of the sizes of the two relations and then applying the joint selectivity of all applicable predicates.

(c) Formulas to estimate the CPU and I/O costs of query execution for every operator. These formulas take into account the statistical properties of its input data streams, existing access methods over the input data streams, and any available order on the data stream (e.g., if a data stream is ordered, then the cost of a sort-merge join on that stream may be significantly reduced). In addition, it is also checked if the output data stream will have any order.

The cost model uses (a)-(c) to compute and associate the following information in a bottom-up fashion for operators in a plan: (1) The size of the data stream represented by the output of the operator node. (2) Any ordering of tuples created or sustained by the output data stream of the operator node. (3) Estimated execution cost for the operator (and the cumulative cost of the partial plan so far).

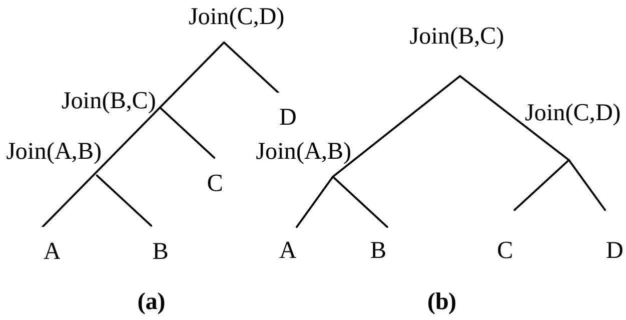

Figure 2. (a) Linear and (b) bushy join

The enumeration algorithm for System-R optimizer demonstrates two important techniques: use of *dynamic programming* and use of *interesting orders*.

The essence of the dynamic programming approach is based on the assumption that the cost model satisfies *the principle of optimality*. Specifically, it assumes that in order to obtain an optimal plan for a SPJ query Q consisting of k joins, it suffices to consider only the optimal plans for subexpressions of Q that consist of (k-1) joins and extend those plans with an additional join. In other words, the suboptimal plans for subexpressions of Q (also called subqueries) consisting of (k-1) joins do not need to be considered further in determining the optimal plan for Q. Accordingly, the dynamic programming based enumeration views a SPJ query `Q` as a *set* of relations $\{R_1, ..R_n\}$ to be joined. The enumeration algorithm proceeds *bottom-up*. At the end of the j-th step, the algorithm produces the optimal plans for all subqueries of size j. To obtain an optimal plan for a subquery consisting of (j+1) relations, we consider all possible ways of constructing a plan for the subquery by extending the plans constructed in the j-th step. For example, the optimal plan for $\{R_1,R_2,R_3,R_4\}$ is obtained by picking the plan with the cheapest cost from among the optimal plans for: (1) $\text{Join}(\{R_1,R_2,R_3\},R_4)$ (2) $\text{Join}(\{R_1,R_2,R_4\},R_3)$ (3) $\text{Join}(\{R_1,R_3,R_4\},R_2)$ (4) $\text{Join}(\{R_2,R_3,R_4\}, R_1)$. The rest of the plans for $\{R_1,R_2,R_3,R_4\}$ may be discarded. The dynamic programming approach is significantly faster than the naïve approach since instead of O(n!) plans, only $O(n2^{n-1})$ plans need to be enumerated.

The second important aspect of System R optimizer is the consideration of *interesting orders*. Let us now consider a query that represents the join among $\{R_1,R_2,R_3\}$ with the predicates $R_1.a = R_2.a = R_3.a$. Let us also assume that the cost of the plans for the subquery $\{R_1,R_2\}$ are x and y for nested-loop and sort-merge join respectively and $x < y$. In such a case, while considering the plan for $\{R_1, R_2, R_3\}$, we will not consider the plan where R_1 and R_2 are joined using sort-merge. However, note that if sort-merge is used to join R_1 and R_2, *the result of the join is sorted on a*. The sorted order may significantly reduce the cost of the join with R_3. Thus, pruning the plan that represents the sort-merge join between R_1 and R_2 can result in sub-optimality of the global plan. The problem arises because the result of the sort-merge join between R_1 and R_2 has an ordering of tuples in the output stream that is useful in the subsequent join. However, the nested-loop join does not have such ordering. Therefore, given a query, System R identified ordering of tuples that are potentially consequential to execution plans for the query (hence the name interesting orders). Furthermore, in the System R optimizer, two plans are compared only if they represent the same expression as well as have the same interesting order. The idea of interesting order was later generalized to *physical properties* in [22] and is used extensively in modern optimizers. Intuitively, a physical property is any characteristic of a plan that is not shared by all plans for the same logical expression, but can impact the cost of subsequent operations. Finally, note that the System-R's approach of taking into account physical properties demonstrates a simple mechanism to handle any violation of the principle of optimality, not necessarily arising only from physical properties.

Despite the elegance of the System-R approach, the framework cannot be easily extended to incorporate other logical transformations (beyond join ordering) that expand the search space. This led to the development of more extensible optimization architectures. However, the use of cost-based optimization, dynamic programming and interesting orders strongly influenced subsequent developments in optimization.

4. SEARCH SPACE

As mentioned in Section 2, the search space for optimization depends on the set of algebraic transformations that preserve equivalence and the set of physical operators supported in an optimizer. In this section, I will discuss a few of the many important algebraic transformations that have been discovered. It should be noted that *transformations do not necessarily reduce cost and therefore must be applied in a cost-based manner by the enumeration algorithm to ensure a positive benefit*.

The optimizer may use several representations of a query during the lifecycle of optimizing a query. The initial representation is often the parse tree of the query and the final representation is an operator tree. An intermediate representation that is also used is that of logical operator trees (also called *query trees*) that captures an algebraic expression. Figure 2 is an example of a query tree. Often, nodes of the query trees are annotated with additional information.

Some systems also use a "calculus-oriented" representation for analyzing the structure of the query. For SPJ queries, such a structure is often captured by a *query graph* where nodes represent relations (correlation variables) and labeled edges represent join predicates among the relations (see Figure 3). Although conceptually simple, such a representation falls short of representing the structure of arbitrary SQL statements in a number of ways. First, predicate graphs only represent a set of join predicates and cannot represent other algebraic operators, e.g., union. Next, unlike natural join, operators such as *outerjoin* are asymmetric and are sensitive to the order of evaluation. Finally, such a representation does not capture the fact that SQL statements may have nested query blocks. In the QGM structure used in the Starburst system [26], the building block is an enhanced query graph that is able to represent a simple SQL statement that has no nesting ("single block" query). Multi block queries are represented as a set of subgraphs with edges among subgraphs that represent predicates (and quantifiers) across query blocks. In contrast, Exodus [22] and its derivatives, uniformly use query trees and operator trees for all phases of optimization.

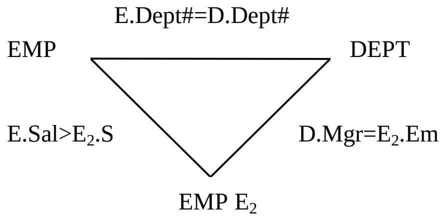

Figure 3. Query

4.1 Commuting Between Operators

A large and important class of transformations exploits commutativity among operators. In this section, we see examples of such transformations.

4.1.1 Generalizing Join Sequencing

In many of the systems, the sequence of join operations is syntactically restricted to limit search space. For example, in the System R project, only linear sequences of join operations are considered and Cartesian product among relations is deferred until after all the joins.

Since join operations are commutative and associative, the sequence of joins in an operator tree need not be linear. In particular, the query consisting of join among relations R_1, R_2, R_3, R_4 can be algebraically represented and evaluated as `Join(Join(A,B),Join(C,D))`. Such query trees are called *bushy*, illustrated in Figure 2(b). Bushy join sequences require materialization of intermediate relations. While bushy trees may result in cheaper query plan, they expand the cost of enumerating the search space considerably[1]. Although there has been some studies of merits of exploring the bushy join sequences, by and large most systems still focus on linear join sequences and only restricted subsets of bushy join trees.

Deferring Cartesian products may also result in poor performance. In many decision-support queries where the query graph forms a star, it has been observed that a Cartesian product among appropriate nodes ("dimensional" tables in OLAP terminology [7]) results in a significant reduction in cost.

In an extensible system, the behavior of the join enumerator may be adapted on a per query basis so as to restrict the "bushy"-ness of the join trees and to allow or disallow Cartesian products [46]. However, it is nontrivial to determine a priori the effects of such tuning on the quality and cost of the search.

4.1.2 Outerjoin and Join

One-sided outerjoin is an asymmetric operator in SQL that preserves all of the tuples of one relation. Symmetric outerjoins preserve both the operand relations. Thus, (R LOJ S), where LOJ designates left outerjoin between R and S, preserves all tuples of R. In addition to the tuples from natural join, the above operation contains all remaining tuples in R that fail to join with S (padded with NULLs for their S attributes). Unlike natural joins, a

[1] It is not the cost of generating the syntactic join orders that is most expensive. Rather, the task of choosing physical operators and computing the cost of each alternative plan is computationally intensive.

sequence of outerjoins and joins do not freely commute. However, when the join predicate is between (R,S) and the outerjoin predicate is between (S,T), the following identity holds:

Join(R, S LOJ T) = Join (R,S) LOJ T

If the above associative rule can be repeatedly applied, we obtain an equivalent expression where evaluation of the "block of joins" precedes the "block of outerjoins". Subsequently, the joins may be freely reordered among themselves. As with other transformations, use of this identity needs to be cost-based. The identities in [53] define a class of queries where joins and outerjoins may be reordered.

4.1.3 Group-By and Join

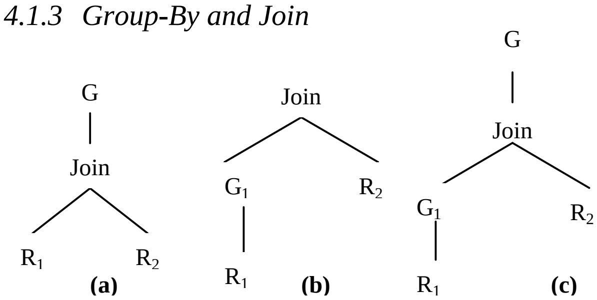

Figure 4. Group By and Join

In traditional execution of a SPJ query with group-by, the evaluation of the SPJ component of the query precedes the group-by. The set of transformations described in this section enable the group by operation to precede a join. These transformations are applicable to queries with `SELECT DISTINCT` since the latter is a special case of group-by. Evaluation of a group-by operator can potentially result in a significant reduction in the number of tuples, since only one tuple is generated for every partition of the relation induced by the group-by operator. Therefore, in some cases, by first doing the group-by, the cost of the join may be significantly reduced. Moreover, in the presence of an appropriate index, a group-by operation may be evaluated inexpensively. A dual of such transformations corresponds to the case where a group-by operator may be pulled up past a join. These transformations are described in [5,60,25,6] (see [4] for an overview).

In this section, we briefly discuss *specific* instances where the transformation to do an early group-by prior to the join may be applicable. Consider the query tree in Figure 4(a). Let the join between R_1 and R_2 be a foreign key join and let the aggregated columns of G be from columns in R_1 and the set of group-by columns be a superset of the foreign key columns of R_1. For such a query, let us consider the corresponding operator tree in Fig. 4(b), where G_1=G. In that tree, the final join with R_2 can only eliminate a set of potential partitions of R_1 created by G_1 but will not affect the partitions nor the aggregates computed for the partitions by G_1 since every tuple in R_1 will join with *at most* one tuple in R_2. Therefore, we can push down the group-by, as shown in Fig. 4(b) and preserve equivalence for *arbitrary* side-effect free aggregate functions. Fig. 4(c) illustrates an example where the transformation *introduces* a group-by and represents a class of useful examples where the group-by operation is done in *stages*. For example, assume that in Fig. 4(a), where all the columns on which aggregated functions are applied are from R_1. In these cases, the introduced group-by operator G_1 partitions the relation on the projection columns of the R_1 node and computes the aggregated values on those partitions. However, the true partitions in Fig 4(a) may need to combine multiple partitions introduced by G_1 into a single partition (many to one mapping). The group-by operator G ensures the above. Such staged computation may still be useful in reducing the cost of the join because of the data reduction effect of G_1. Such staged aggregation requires the aggregating function to satisfy the property that `Agg(S U S')` can be computed from `Agg(S)` and `Agg(S')`. For example, in order to compute total sales for all products in each division, we can use the transformation in Fig. 4(c) to do an early aggregation and obtain the total sales for each product. We then need a subsequent group-by that sums over all products that belong to each division.

4.2 Reducing Multi-Block Queries to Single-Block

The technique described in this section shows how under some conditions, it is possible to collapse a multi-block SQL query into a single block SQL query.

4.2.1 Merging Views

Let us consider a conjunctive query using SELECT ANY. If one or more relations in the query are views, but each is defined through a conjunctive query, then the view definitions can simply be "unfolded" to obtain a single block SQL query. For example, if a query `Q = Join(R,V)` and view `V = Join(S,T)`, then the query Q can be unfolded to `Join(R,Join(S,T))` and may be freely reordered. Such a step may require some renaming of the variables in the view definitions.

Unfortunately, this simple unfolding fails to work when the views are more complex than simple SPJ queries. When one or more of the views contain `SELECT DISTINCT`, transformations to move or pull up `DISTINCT` need to be careful to preserve the number of duplicates correctly, [49]. More generally, when the view contains a group by operator, unfolding requires the ability to pull-up the group-by operator and then to freely reorder not only the joins but also the group-by operator to ensure optimality. In particular, we are given a query such as the one in Fig. 4(b) and we are trying to consider how we can transform it in a form such as Fig. 4(a) so that R_1 and R_2 may be freely reordered. While the transformations in Section 4.1.3 may be used in such cases, it underscores the complexity of the problem [6].

4.2.2 Merging Nested Subqueries

Consider the following example of a nested query from [13] where `Emp#` and `Dept#` are keys of the corresponding relations:

```
SELECT Emp.Name
FROM Emp
WHERE Emp.Dept#  IN
      SELECT Dept.Dept#  FROM Dept
      WHERE Dept.Loc='Denver'
       AND Emp.Emp# = Dept.Mgr
```

If tuple iteration semantics are used to answer the query, then the inner query is evaluated for each tuple of the Dept relation once. An obvious optimization applies when the inner query block contains no variables from the outer query block (*uncorrelated*). In such cases, the inner query block needs to be evaluated only once. However, when there is indeed a variable from the outer block, we say that the query blocks are correlated. For example, in the query above, `Emp.Emp#` acts as the correlated variable. Kim [35] and subsequently others [16,13,44] have identified techniques to unnest a correlated nested SQL query and "flatten" it to a single query. For example, the above nested query reduces to:

```
SELECT E.Name
FROM Emp E, Dept D
WHERE E.Dept# = D.Dept#
AND D.Loc = 'Denver' AND E.Emp# = D.Mgr
```

Dayal [13] was the first to offer an *algebraic* view of unnesting. The complexity of the problem depends on the structure of the nesting, i.e., whether the nested subquery has quantifiers (e.g., `ALL`, `EXISTS`), aggregates or neither. In the simplest case, of which the above query is an example, [13] observed that the tuple semantics can be modeled as `Semijoin(Emp,Dept, Emp.Dept# = Dept.Dept#)`[2]. Once viewed this way, it is not hard to see why the query may be merged since:

```
Semijoin(Emp,Dept,Emp.Dept# = Dept. Dept#) =
Project(Join(Emp,Dept), Emp.*)
```

Where `Join(Emp,Dept)` is on the predicate `Emp.Dept# = Dept. Dept#`. The second argument of the Project operator[3] indicates that all columns of the relation Emp must be retained.

The problem is more complex when aggregates are present in the nested subquery, as in the example below from [44] since merging query blocks now requires pulling up the aggregation without violating the semantics of the nested query:

```
SELECT Dept.name
FROM Dept
WHERE Dept.num-of-machines ≥
(SELECT COUNT(Emp.*) FROM Emp
 WHERE Dept.name= Emp.Dept_name)
```

It is especially tricky to preserve duplicates and nulls. To appreciate the subtlety, observe that if for a specific value of `Dept.name` (say d), there are no tuples with a matching `Emp.Dept_name`, i.e., even if the predicate `Dept.name= Emp.dept_name` fails, then there is still an output tuple for the `Dept` tuple `d`. However, if we were to adopt the transformation used in the first query of this section, then there will be no output tuple for the `dept d` since the join predicate fails. Therefore, in the presence of aggregation, we must preserve all the tuples of the outer query block by a left *outerjoin*. In particular, the above query can be correctly transformed to:

```
SELECT Dept.name    FROM Dept LEFT OUTER JOIN Emp
ON (Dept.name= Emp.dept_name )
GROUP BY Dept.name
HAVING Dept. num-of-machines < COUNT (Emp.*)
```

Thus, for this class of queries the merged single block query has outerjoins. If the nesting structure among query blocks is linear, then this approach is applicable and transformations produce a single block query that consists of a linear sequence of joins and outerjoins. It turns out that the sequence of joins and outerjoins is such that we can use the associative rule described in Section 4.1.2 to compute all the joins first and then do all the outerjoins in sequence. Another approach to unnesting subqueries is to transform a query into one that uses table-expressions or views (and therefore, *not* a single block query). This was the direction of Kim's work [35] and it was subsequently refined in [44].

4.3 Using Semijoin Like Techniques for Optimizing Multi-Block Queries

In the previous section, I presented examples of how multi-block queries may be collapsed in a single block. In this section, I discuss a complementary approach. The goal of the approach described in this section is to exploit the selectivity of predicates across blocks.[4] It is conceptually similar to the idea of using semijoin to propagate from a site A to a remote site B information on relevant values of A so that B sends to A no unnecessary tuples. In the context of multi-block queries, A and B are in different query blocks but are parts of the same query and therefore the transmission cost is not an issue. Rather, the information "received from A" is used to reduce the computation needed in B as well as to ensure that the results produced by B are relevant to A as well. This technique requires introducing new table expressions and views. For example, consider the following query from [56]:

```
CREATE VIEW DepAvgSal As (
       SELECT  E.did,  Avg(E.Sal)  AS  avgsal
       FROM Emp E
       GROUP BY E.did)
SELECT E.eid, E.sal
FROM Emp E, Dept D, DepAvgSal V
WHERE  E.did   =  D.did  AND  E.did  =  V.did
AND E.age < 30 AND D.budget > 100k
AND E.sal > V.avgsal
```

The technique recognizes that we can create the set of relevant `E.did` by doing only the join between `E` and `D` in the above query and projecting the unique `E.did`. This set can be passed to the view `DepAvgSal` to restrict its computation. This is accomplished by the following three views.

```
CREATE VIEW partialresult AS
(SELECT E.id, E.sal, E.did
 FROM Emp E, Dept D
 WHERE E.did=D.did AND E.age < 30
 AND D.budget > 100k)
CREATE VIEW Filter AS
(SELECT DISTINCT P.did FROM PartialResult P)
CREATE VIEW LimitedAvgSal AS
(SELECT E.did, Avg(E.Sal) AS avgsal
FROM Emp E, Filter F
WHERE E.did = F.did GROUP BY E.did)
```

The reformulated query on the next page exploits the above views to restrict computation.

---

[2] Semijoin(A,B,P) stands for semijoin between A and B that preserves attributes of A and where P is the semijoin predicate.

[3] I assume that the operator does not remove duplicates.

[4] Although this technique historically developed as a derivative of Magic Sets and sideways information passing [2], I find the relationship to semijoin more intuitive and less magical.

```
SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

The above technique can be used in a multi-block query containing view (including recursive view) definitions or nested subqueries [42,43,56,57]. In each case, the goal is to avoid redundant computation in the views or the nested subqueries. It is also important to recognize the tradeoff between the cost of computing the views (the view `PartialResult` in the example above) and use of such views to reduce the cost of computation.

The formal relationship of the above transformation to semijoin has recently been presented in [56] and may form the basis for integration of this strategy in a cost-based optimizer. Note that a degenerate application of this technique is passing the predicates across query blocks instead of results of views. This simpler technique has been used in distributed and heterogeneous databases and generalized in [36].

5. STATISTICS AND COST ESTIMATION

Given a query, there are many logically equivalent algebraic expressions and for each of the expressions, there are many ways to implement them as operators. Even if we ignore the computational complexity of enumerating the space of possibilities, there remains the question of deciding which of the operator trees consumes the least resources. Resources may be CPU time, I/O cost, memory, communication bandwidth, or a combination of these. Therefore, given an operator tree (partial or complete) of a query, being able to *accurately* and *efficiently* evaluate its cost is of fundamental importance. The cost estimation must be accurate because *optimization is only as good as its cost estimates*. Cost estimation must be efficient since it is in the inner loop of query optimization and is repeatedly invoked. The basic estimation framework is derived from the System-R approach:

1. Collect statistical summaries of data that has been stored.
2. Given an operator and the statistical summary for each of its input data streams, determine the:
   (a) Statistical summary of the output data stream
   (b) Estimated cost of executing the operation

Step 2 can be applied iteratively to an operator tree of arbitrary depth to derive the costs for each of its operators. Once we have the costs for each of the operator nodes, the cost for the plan may be obtained by combining the costs of each of the operator nodes in the tree. In Section 5.1, we discuss the statistical parameters for the stored data that are used in cost optimization and efficient ways of obtaining such statistical information. We also discuss how to propagate such statistical information. The issue of estimating cost for physical operators is discussed in Section 5.2.

It is important to recognize the differences between the nature of the statistical property and the cost of a plan. The statistical property of the output data stream of a plan is the same as that of any other plan for the same query, but its cost can be different from other plans. In other words, statistical summary is a logical property but the cost of a plan is a physical property.

5.1 Statistical Summaries of Data

5.1.1 Statistical Information on Base Data

For every table, the necessary statistical information includes the number of tuples in a data stream since this parameter determines the cost of data scans, joins, and their memory requirements. In addition to the number of tuples, the number of physical pages used by the table is important. Statistical information on columns of the data stream is of interest since these statistics can be used to estimate the selectivity of predicates on that column. Such information is created for columns on which there are one or more indexes, although it may be created on demand for any other column as well.

In a large number of systems, information on the data distribution on a column is provided by *histograms*. A histogram divides the values on a column into k buckets. In many cases, k is a constant and determines the degree of accuracy of the histogram. However, k also determines the memory usage, since while optimizing a query, relevant columns of the histogram are loaded in memory. There are several choices for "bucketization" of values. In many database systems, equi-depth (also called equi-height) histograms are used to represent the data distribution on a column. If the table has n records and the histogram has k buckets, then an equi-depth histogram divides the set of values on that column into k ranges such that each range has the same *number* of values, i.e., n/k. Compressed histograms place frequently occurring values in singleton buckets. The number of such singleton buckets may be tuned. It has been shown in [52] that such histograms are effective for either high or low skew data. One aspect of histograms relevant to optimization is the assumption made about values within a bucket. For example, in an equi-depth histogram, values within the endpoints of a bucket may be assumed to occur with uniform spread. A discussion of the above assumption as well as a broad taxonomy of histograms and ramifications of the histogram structures on accuracy appears in [52]. In the absence of histograms, information such as the *min* and *max* of the values in a column may be used. However, in practice, the second lowest and the second highest values are used since the *min* and *max* have a high probability of being outlying values. Histogram information is complemented by information on parameters such as number of distinct values on that column

Although histograms provide information on a single column, they do not provide information on the *correlations* among columns. In order to capture correlations, we need the *joint* distribution of values. One option is to consider 2-dimensional histograms [45,51]. Unfortunately, the space of possibilities is quite large. In many systems, instead of providing detailed joint distribution, only summary information such as the number of distinct pairs of values is used. For example, the statistical information associated with a multi-column index may consist of a histogram on the leading column and the total count of distinct combinations of column values present in the data.

5.1.2 Estimating Statistics on Base Data

Enterprise class databases often have large schema and also have large volumes of data. Therefore, to have the flexibility of obtaining statistics to improve accuracy, it is important to be able to estimate the statistical parameters accurately and efficiently. *Sampling* data provides one possible approach. However, the challenge is to limit the error in estimation. In [48], Shapiro and Connell show that for a *given* query, only a small sample is needed to estimate a histogram that has a high probability of being accurate *for the given query*. However, this misses the point since the goal is to build a histogram that is reasonably accurate for a *large class of queries*. Our recent work has addressed this problem [11]. We have also shown that the task of estimating distinct values is *provably* error prone, i.e., for any estimation scheme, there exists a database where the error is significant. This result explains the past difficulty in estimation of the number of distinct values [50,27]. Recent work has also addressed the problem of maintaining statistics in an incremental fashion [18].

5.1.3 Propagation of Statistical Information

It is not sufficient to use information only on base data because a query typically contains many operators. Therefore, it is important to be able to propagate the statistical information through operators. The simplest case of such an operator is selection. If there is a histogram on a column A and the query is a simple selection on column A, then the histogram can be modified to reflect the effect of the selection. Some inaccuracy results in this step due to assumptions such as uniform spread that needs to be made within a bucket. Moreover, the inability to capture correlation is a key source of error. In the above example, this will be reflected in *not* modifying the distribution of other attributes on the table (except A) and thus incurring potentially significant errors in subsequent operators. Likewise, if multiple predicates are present, then the independence assumption is made and the product of the selectivity is considered. However, some systems only use the selectivity of the most selective predicate and can also identify potential correlation [17]. In the presence of histograms on columns involved in a join predicate, the histograms may be "joined". However, this raises the issue of aligning the corresponding buckets. Finally, when histogram information is not available, then ad-hoc constants are used to estimate selectivity, as in [55].

5.2 Cost Computation

The cost estimation step tries to determine the cost of an operation. The costing module estimates CPU, I/O and, in the case of parallel or distributed systems, communication costs. In most systems, these parameters are combined into an overall metric that is used for comparing alternative plans. The problem of choosing an appropriate set of to determine cost requires considerable care. An early study [40] identified that in addition to the physical and statistical properties of the input data streams and the computation of selectivity, modeling buffer utilization plays a key role in accurate estimation. This requires using different buffer pool hit ratios depending on the levels of indexes as well as adjusting buffer utilization by taking into account properties of join methods, e.g., a relatively pronounced locality of reference in an index scan for indexed nested loop join [17]. Cost models take into account relevant aspects of physical design, e.g., co-location of data and index pages. However, the ability to do accurate cost estimation and propagation of statistical information on data streams remains one of the difficult open issues in query optimization.

6. ENUMERATION ARCHITECTURES

An enumeration algorithm must pick an inexpensive execution plan for a given query by exploring the search space. The System-R join enumerator that we discussed in Section 3 was designed to choose only an optimal linear join order. A software engineering consideration is to build the enumerator so that it can gracefully adapt to changes in the search space due to the addition of new transformations, the addition of new physical operators (e.g., a new join implementation) and changes in the cost estimation techniques. More recent optimization architectures have been built with this paradigm and are called *extensible* optimizers. Building an extensible optimizer is a tall order since it is more than simply coming up with a better enumeration algorithm. Rather, they provide an infrastructure for evolution of optimizer design. However, generality in the architecture must be balanced with the need for efficiency in enumeration.

We focus on two representative examples of such extensible optimizers: *Starburst* and *Volcano*/*Cascades* briefly. Despite their differences, we can summarize some of the commonality in them: (a) Use of generalized cost functions and physical properties with operator nodes. (b) Use of a rule engine that allows transformations to modify the query expression or the operator trees. Such rule engines also provide the ability to direct search to achieve efficiency. (c) Many exposed "knobs" that can be used to tune the behavior of the system. Unfortunately, setting these knobs for optimal performance is a daunting task

6.1 Starburst

Query optimization in the Starburst project [26] at IBM Almaden begins with a structural representation of the SQL query that is used throughout the lifecycle of optimization. This representation is called the Query Graph Model (QGM). In the QGM, a *box* represents a query block and labeled arcs between boxes represent table references across blocks. Each box contains information on the predicate structure as well as on whether the data stream is ordered. In the *query rewrite* phase of optimization [49], rules are used to transform a QGM into another equivalent QGM. Rules are modeled as pairs of arbitrary functions. The first one checks the condition for applicability and the second one enforces the transformation. A forward chaining rule engine governs the rules. Rules may be grouped in rule classes and it is possible to tune the order of evaluation of rule classes to focus search. Since any application of a rule results in a valid QGM, any set of rule applications guarantee query equivalence (assuming rules themselves are valid). The query rewrite phase does not have the cost information available. This forces this module to either retain alternatives obtained through rule application or to use the rules in a heuristic way (and thus compromise optimality).

The second phase of query optimization is called *plan optimization*. In this phase, given a QGM, an execution plan (operator tree) is chosen. In Starburst, the physical operators (called LOLEPOPs) may be combined in a variety of ways to implement higher level operators. In Starburst, such combinations are expressed in a grammar production-like language [37]. The realization of a higher-level operation is expressed by its derivation in terms of the physical operators. In computing such derivations, comparable plans that represent the same physical and logical properties but have higher costs, are pruned. Each plan has a *relational* description that corresponds to the algebraic expression it represents, an *estimated* cost, and *physical* properties (e.g., order). These properties are propagated as plans are built bottom-up. Thus, with each physical operator, a function is associated that shows the effect of the physical operator on each of the above properties. The join enumerator in this system is similar to System-R's bottom-up enumeration scheme.

6.2 Volcano/Cascades

The Volcano [23] and the Cascades extensible architecture [21] evolved from Exodus [22]. In these systems, rules are used universally to represent the knowledge of search space. Two kinds of rules are used. The transformation rules map an algebraic expression into another. The implementation rules map an algebraic expression into an operator tree. The rules may have conditions for applicability. Logical properties, physical properties and costs are associated with plans. The physical properties and the cost depend on the algorithms used to implement operators and its input data streams. For efficiency, Volcano/Cascades uses dynamic programming in a top-down way ("memoization"). When presented with an optimization task, it checks whether the task has already been accomplished by looking up its logical and physical properties in the table of plans that have been optimized in the past. Otherwise, it will apply a logical transformation rule, an implementation rule, or use an enforcer to modify properties of the data stream. At every stage, it uses the *promise* of an action to determine the next move. The promise parameter is programmable and reflects cost parameters.

The Volcano/Cascades framework differs from Starburst in its approach to enumeration: (a) These systems do not use two distinct optimization phases because all transformations are algebraic and cost-based. (b) The mapping from algebraic to physical operators occurs in a single step. (c) Instead of applying rules in a forward chaining fashion, as in the Starburst query rewrite phase, Volcano/Cascades does goal-driven application of rules.

7. BEYOND THE FUNDAMENTALS

So far I have covered the basics of the software components of the optimizer. In this section, I discuss some of the more advanced issues. Each of these issues is of considerable importance in commercial systems.

7.1 Distributed and Parallel Databases

Distributed databases introduce issues of communication costs and an expanded search space due to the fact that it is possible to move data and choose sites for intermediate operations in optimizing a query. While some of the early work focused almost exclusively on reducing communication costs [1,3] (e.g., using semijoins), results from System R* pointed out the dominant role of local processing [39] (See [38] for an overview). Over time, distributed database architectures have evolved into either replicated databases to handle physical distribution or to parallel databases for scale-up. In replicated architectures, maintaining consistency across replicas is an important issue but is beyond the scope of this article.

Unlike distributed systems, *Parallel* databases behave as a single system but exploit multiple processing elements to reduce the *response time*[5] of queries. Benefits of parallelism may be harnessed in several ways. For example, physical data distribution, where a table (in general, a data stream) is partitioned or replicated among nodes, enables processors to work on independent data sets. Parallelism may also be harnessed for independent operation or pipelined operation (by placing the producer and the consumer nodes on different processors). The advantages of parallelism are counteracted by the need for communication among the processors to exchange data, e.g., when data needs to be *repartitioned* after an operation. Furthermore, effective *scheduling* of physical operators on processors brings a new dimension to the optimization problem. The XPRS project [31,32] advocated a two-phase approach where traditional single processor query optimization is used to generate an execution plan in the first phase. In the second phase, scheduling of processors was determined. The query optimization work in XPRS did not study the effects of processor communication. Work by Hasan [28] demonstrated the importance of taking communication costs into account. Hasan retained the two-phase optimization framework used in XPRS, but incorporated the cost and benefits of data repartitioning in the first phase of the optimization to determine the join order and the access methods used. The partitioning attribute of a data stream was treated as a physical property of the data stream. The output of the first phase is a physical operator tree that exposes precedence constraints (e.g., for sorting) and pipelined execution. For the second phase of optimization, he proposed scheduling algorithms that take into account communication costs [28].

7.2 User-Defined Functions

Stored procedures (also called user-defined functions) have become widely available in relational systems. While their support varies from one product to another, they provide a powerful mechanism to reduce client-server communication and provide the means for incorporating application semantics in querying. New optimization questions arise when such stored procedures are treated as first-order citizens in the query. The problem of determining the cost model of user-defined functions remains a difficult problem. Interesting issues also arise in the context of the enumeration algorithm. For example, consider the case when the stored procedure acts as a user-defined predicate in the WHERE clause of a query. Unlike other predicates, such predicates may be expensive (e.g., since they may be predicates on a BLOB such as an image) and therefore it is no longer a sound heuristic to evaluate such predicates as early as possible. The problem of optimizing queries with user-defined predicates was posed in [12,29]. The approach taken in [12] has been to treat the user-defined predicates as a relation from the point of view of dynamic query optimization. The approaches in [29,30] exploit the observation that if there are no joins, then the expensive predicates may be ordered efficiently by their *ranks*, computed from their selectivity and per tuple cost of evaluation. Unfortunately, their attempt to extend the use of ranks for queries with joins may result in suboptimal plans. This shortcoming is resolved in [8] by representing the application of user-defined predicates like a physical property of a plan so that the dynamic-programming-based enumeration algorithm guarantees optimality. Moreover, for realistic assumptions of the cost model, it is shown that the problem is polynomial in the number of user-defined predicates.

Solving the problem of optimizing user-defined predicates is only a first step in the broader problem of representing the semantics of ADTs in a query system and optimizing queries over ADTs. This problem is also intimately tied to the area of semantic query optimization.

---

[5] In single processor systems, the key focus has been reducing total work. Parallel execution attempts to reduce response rather than work. Indeed, in many cases , although not necessarily, parallel execution increases total work.

7.3 Materialized Views

Materialized views are results of views (i.e., queries) that are cached by the querying subsystem and used by the optimizer transparently. The optimization problem is as follows: Given a set of materialized views and a query, the goal is to optimize the query while taking into account the materialized views that are present. This problem introduces two fundamental challenges. First, the problem of reformulating the query so that it can use one or more of the materialized views must be addressed. The general problem is undecidable and even determining effective sufficient conditions are nontrivial due to the complexity of SQL. This problem has been addressed in the context of single block SQL queries only [15,61,59,9] and must be extended for complex queries. Next, approaching the optimization problem as a two step process, where all logically equivalent expressions are generated and each one of them is individually optimized, may increase the cost of optimization since subexpressions are not pruned in a cost-based way. In [9], we showed how the steps of enumerating and generating equivalent expressions in the presence of materialized views may be overlapped.

7.4 Other Optimization Issues

In this paper, I have been able to touch only on some of the foundational issues in query optimization. There are many important areas that I have not discussed. One interesting direction is that of being able to defer generation of complete plans subject to availability of runtime information [19,33]. Also, the problem of considering other resources, especially memory, in determining execution plans remains an open issue. The work in [58] addresses the issue of optimizing the use of order in query optimization. Optimizer technology in Object-Oriented Systems is an important area that is worthy of a separate discussion. Furthermore, as database systems get used in multimedia and web context, being able to address fuzzy (imprecise) queries is an interesting direction of work [14,10]. Recent emphasis on decision support systems has also sparked work in SQL extensions. Such work, as in CUBE [24], is not motivated by the need for expressive power, but rather seeks to extend the language so that the optimizer can use the constructs to optimize decision support systems better.

8. CONCLUSION

Optimization is much more than transformations and query equivalence. The infrastructure for optimization is significant. Designing effective and correct SQL transformations is hard, developing a robust cost metric is elusive, and building an extensible enumeration architecture is a significant undertaking. Despite many years of work, significant open problems remain. However, an understanding of the existing engineering framework is necessary for making effective contribution to the area of query optimization.

ACKNOWLEDGMENTS

My many informal discussions with Umesh Dayal, Goetz Graefe, Waqar Hasan, Ravi Krishnamurthy, Guy Lohman, Hamid Pirahesh, Kyuseok Shim and Jeff Ullman have greatly helped me develop my understanding of SQL optimization. Many of them also helped me improve this draft through their comments. I am also very grateful to Latha Colby, William McKenna, Vivek Narasayya, and Janet Wiener for their insightful comments on the draft. As always, my thanks to Debjani for her patience.

9. REFERENCES

[1] Apers, P.M.G., Hevner, A.R., Yao, S.B. Optimization Algorithms for Distributed Queries. IEEE Transactions on Software Engineering, Vol 9:1, 1983.

[2] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D. Magic sets and other strange ways to execute logic programs. In Proc. of ACM PODS, 1986.

[3] Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L, Rothnie, J. Query Processing in a System for Distributed Databases (SDD-1), ACM TODS 6:4 (Dec 1981).

[4] Chaudhuri, S., Shim K. An Overview of Cost-based Optimization of Queries with Aggregates. IEEE DE Bulletin, Sep. 1995. (Special Issue on Query Processing).

[5] Chaudhuri, S., Shim K. Including Group-By in Query Optimization. In Proc. of VLDB, Santiago, 1994.

[6] Chaudhuri, S., Shim K. Query Optimization with aggregate views: In Proc. of EDBT, Avignon, 1996.

[7] Chaudhuri, S., Dayal, U. An Overview of Data Warehousing and OLAP Technology. In ACM SIGMOD Record, March 1997.

[8] Chaudhuri, S., Shim K. Optimization of Queries with User-defined Predicates. In Proc. of VLDB, Mumbai, 1996.

[9] Chaudhuri, S., Krishnamurthy, R., Potamianos, S., Shim K. Optimizing Queries with Materialized Views. In Proc. of IEEE Data Engineering Conference, Taipei, 1995.

[10] Chaudhuri, S., Gravano, L. Optimizing Queries over Multimedia Repositories. In Proc. of ACM SIGMOD, Montreal, 1996.

[11] Chaudhuri, S., Motwani, R., Narasayya, V. Random Sampling for Histogram Construction: How much is enough? In Proc. of ACM SIGMOD, Seattle, 1998.

[12] Chimenti D., Gamboa R., Krishnamurthy R. Towards an Open Architecture for LDL. In Proc. of VLDB, Amsterdam, 1989.

[13] Dayal, U. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers. In Proc. of VLDB, 1987.

[14] Fagin, R. Combining Fuzzy Information from Multiple Systems. In Proc. of ACM PODS, 1996.

[15] Finkelstein S., Common Expression Analysis in Database Applications. In Proc. of ACM SIGMOD, Orlando, 1982.

[16] Ganski, R.A., Long, H.K.T. Optimization of Nested SQL Queries Revisited. In Proc. of ACM SIGMOD, San Francisco, 1987.

[17] Gassner, P., Lohman, G., Schiefer, K.B. Query Optimization in the IBM DB2 Family. IEEE Data Engineering Bulletin, Dec. 1993.

[18] Gibbons, P.B., Matias, Y., Poosala, V. Fast Incremental Maintenance of Approximate Histograms. In Proc. of VLDB, Athens, 1997.

[19] Graefe, G., Ward K. Dynamic Query Evaluation Plans. In Proc. of ACM SIGMOD, Portland, 1989.

[20] Graefe G. Query Evaluation Techniques for Large Databases. In ACM Computing Surveys: Vol 25, No 2., June 1993.

[21] Graefe, G. The Cascades Framework for Query Optimization. In Data Engineering Bulletin. Sept. 1995.

[22] Graefe, G., Dewitt D.J. The Exodus Optimizer Generator. In Proc. of ACM SIGMOD, San Francisco, 1987.

[23] Graefe, G., McKenna, W.J. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Proc. of the IEEE Conference on Data Engineering, Vienna, 1993.

[24] Gray, J., Bosworth, A., Layman A., Pirahesh H. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub-Totals. In Proc. of IEEE Conference on Data Engineering, New Orleans, 1996.

[25] Gupta A., Harinarayan V., Quass D. Aggregate-query processing in data warehousing environments. In Proc. of VLDB, Zurich, 1995.

[26] Haas, L., Freytag, J.C., Lohman, G.M., Pirahesh, H. Extensible Query Processing in Starburst. In Proc. of ACM SIGMOD, Portland, 1989.

[27] Haas, P.J., Naughton, J.F., Seshadri, S., Stokes, L. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In Proc. of VLDB, Zurich, 1995.

[28] Hasan, W. Optimization of SQL Queries for Parallel Machines. LNCS 1182, Springer-Verlag, 1996.

[29] Hellerstein J.M., Stonebraker, M. Predicate Migration: Optimization queries with expensive predicates. In Proc. of ACM SIGMOD, Washington D.C., 1993.

[30] Hellerstein, J.M. Predicate Migration placement. In Proc. of ACM SIGMOD, Minneapolis, 1994.

[31] Hong, W., Stonebraker, M. Optimization of Parallel Query Execution Plans in XPRS. In Proc. of Conference on Parallel and Distributed Information Systems. 1991.

[32] Hong, W. Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays. Ph.D. Thesis, University of California, Berkeley, 1992.

[33] Ioannidis, Y., Ng, R.T., Shim, K., Sellis, T. Parametric Query Optimization. In Proc. of VLDB, Vancouver, 1992.

[34] Ioannidis, Y.E. Universality of Serial Histograms. In Proc. of VLDB, Dublin, Ireland, 1993.

[35] Kim, W. On Optimizing an SQL-like Nested Query. ACM TODS, Vol 9, No. 3, 1982.

[36] Levy, A., Mumick, I.S., Sagiv, Y. Query Optimization by Predicate Move-Around. In Proc. of VLDB, Santiago, 1994.

[37] Lohman, G.M. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In Proc. of ACM SIGMOD, 1988.

[38] Lohman. G., Mohan, C., Haas, L., Daniels, D., Lindsay, B., Selinger, P., Wilms, P. Query Processing in R*. In Query Processing in Database Systems. Springer Verlag, 1985.

[39] Mackert, L.F., Lohman, G.M. R* Optimizer Validation and Performance Evaluation For Distributed Queries. In Readings in Database Systems. Morgan Kaufman.

[40] Mackert, L.F., Lohman, G.M. R* Optimizer Validation and Performance Evaluation for Local Queries. In Proc. of ACM SIGMOD, 1986.

[41] Melton, J., Simon A. Understanding The New SQL: A Complete Guide. Morgan Kaufman.

[42] Mumick, I.S., Finkelstein, S., Pirahesh, H., Ramakrishnan, R. Magic is Relevant. In Proc. of ACM SIGMOD, Atlantic City, 1990.

[43] Mumick, I.S., Pirahesh, H. Implementation of Magic Sets in a Relational Database System. In Proc. of ACM SIGMOD, Montreal, 1994.

[44] Muralikrishna, M. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In Proc. of VLDB, Vancouver, 1992.

[45] Muralikrishna M., Dewitt D.J. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries, Proc. of ACM SIGMOD, Chicago, 1988.

[46] Ono, K., Lohman, G.M. Measuring the Complexity of Join Enumeration in Query Optimization. In Proc. of VLDB, Brisbane, 1990.

[47] Ozsu M.T., Valduriez, P. Principles of Distributed Database Systems. Prentice-Hall, 1991.

[48] Piatetsky-Shapiro, G., Connell, C. Accurate Estimation of the Number of Tuples Satisfying a Condition. In Proc. of ACM SIGMOD, 1984.

[49] Pirahesh, H., Hellerstein J.M., Hasan, W. Extensible/Rule Based Query Rewrite Optimization in Starburst. In Proc. of ACM SIGMOD 1992.

[50] Poosala, V., Ioannidis, Y., Haas, P., Shekita, E. Improved Histograms for Selectivity Estimation. In Proc. of ACM SIGMOD, Montreal, Canada 1996.

[51] Poosala, V., Ioannidis, Y.E. Selectivity Estimation Without the Attribute Value Independence Assumption. In Proc. of VLDB, Athens, 1997.

[52] Poosala, V., Ioannidis, Y.E., Haas, P.J., Shekita, E.J. Improved Histograms for Selectivity Estimation of Range Predicates In Proc. of ACM SIGMOD, Montreal, 1996.

[53] Rosenthal, A., Galindo-Legaria, C. Query Graphs, Implementing Trees, and Freely Reorderable Outerjoins. In Proc. of ACM SIGMOD, Atlantic City, 1990.

[54] Schneider, D.A. Complex Query Processing in Multiprocessor Database Machines. Ph.D. thesis, University of Wisconsin, Madison, Sept. 1990. Computer Sciences Technical Report 965.

[55] Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price T.G. Access Path Selection in a Relational Database System. In Readings in Database Systems. Morgan Kaufman.

[56] Seshadri P., et al. Cost Based Optimization for Magic: Algebra and Implementation. In Proc. of ACM SIGMOD, Montreal, 1996.

[57] Seshadri, P., Pirahesh, H., Leung, T.Y.C. Decorrelating complex queries. In Proc. of the IEEE International Conference on Data Engineering, 1996.

[58] Simmen, D., Shekita E., Malkemus T. Fundamental Techniques for Order Optimization. In Proc. of ACM SIGMOD, Montreal, 1996.

[59] Srivastava D., Dar S., Jagadish H.V., Levy A.: Answering Queries with Aggregation Using Views. Proc. of VLDB, Mumbai, 1996.

[60] Yan, Y.P., Larson P.A. Eager aggregation and lazy aggregation. In Proc. of VLDB Conference, Zurich, 1995.

[61] Yang, H.Z., Larson P.A. Query Transformation for PSJ-Queries. In Proc. of VLDB, 1987.