Selectivity-Based Partitioning: A Divide-and-Union Paradigm For Effective Query Optimization

Neoklis Polyzotis
Univ. of California Santa Cruz
alkis@cs.ucsc.edu

ABSTRACT

Modern query optimizers select an efficient join ordering for a physical execution plan based essentially on the *average* join selectivity factors among the referenced tables. In this paper, we argue that this "monolithic" approach can miss important opportunities for the effective optimization of relational queries. We propose *selectivity-based partitioning*, a novel optimization paradigm that takes into account the join correlations among *relation fragments* in order to essentially enable multiple (and more effective) join orders for the evaluation of a single query. In a nutshell, the basic idea is to carefully partition a relation according to the selectivities of the join operations, and subsequently rewrite the query as a union of constituent queries over the computed partitions. We provide a formal definition of the related optimization problem and derive properties that characterize the set of optimal solutions. Based on our analysis, we develop a heuristic algorithm for computing efficiently an effective partitioning of the input query. Results from a preliminary experimental study verify the effectiveness of the proposed approach and demonstrate its potential as an effective optimization technique.

Categories and Subject Descriptors: H.2.4 [Database Management]: Query Processing, Relational Databases

General Terms: Algorithms, Performance

1. INTRODUCTION

Effective query optimization techniques have played a key role in the success of relational database systems as they have enabled the efficient evaluation of high-level, declarative queries over massive data stores. At an abstract level, the outcome of query optimization is a low-cost physical execution plan for accessing the stored data and computing the results of the input declarative query. The complexity of the problem clearly makes this a challenging task, and the query optimizer has naturally evolved to one of the most complex (and most important) modules of a modern DBMS.

A central issue in relational query optimization remains the selection of an effective join ordering, i.e., an order for evaluating efficiently the join predicates of a given query. This important problem has been the focus of active research from the first years of relational database development and earlier studies have introduced a host of effective optimization techniques [3, 9, 13, 17, 18, 19]. At an abstract level, the proposed methods follow the same basic approach of exploring the joint space of join orders and their implementations in physical plans, and selecting the one with the least estimated cost. Typically, the cost factors of a candidate plan depend on the average selectivity of the involved join operators, that is, the total number of tuples that are generated when the corresponding tables are joined. An effective plan intuitively minimizes the number of intermediate results, thus reducing the amount of work that is required to produce the answer to a query.

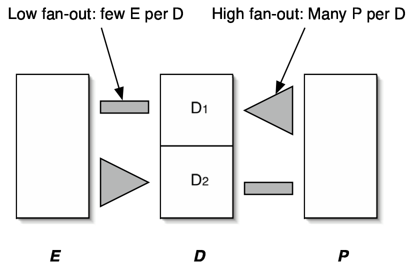

Figure 1: Statistical correlation among join attributes in $E \bowtie D \bowtie P$.

A key observation on existing optimization techniques is that they adopt a "monolithic" approach to the problem of join ordering: for any query, the optimizer selects a *single* join order that is based on the *average* join selectivities among different tables. In effect, the optimized physical plan depends on the average statistical profile of the data and essentially ignores the details of individual frequency distributions. To illustrate this point, consider a simple database schema consisting of three tables, namely, *E*mployee, *D*epartment, and *P*roject, and assume that E and P have foreign key relationships to D. Let us assume that these two relationships are correlated in the following manner: either a department has a few big-budget projects that employ a large number of employees, or it has many small budget projects and a few employees to work on them. This correlation is shown pictorially in Figure 1, where D can be separated in two partitions, D_1 and D_2, according to the 'fan-out' of department tuples to tables E and P. Consider now the query $E \bowtie D \bowtie P$, that pairs each employee with the projects he/she works on. A conventional query optimizer will base its decisions on the overall selectivity factors for the joins $E \bowtie D$ and $D \bowtie P$, which essentially assign the same average fan-out factor to every tuple in D even if the detailed statistical characteristics are very different.

In this paper, we argue that this monolithic approach can miss important opportunities for optimizing queries more effectively. Returning to our example, consider the option of partitioning D to D_1 and D_2, based on the statistical dependencies of these partitions to the other relations, and essentially rewriting the original query $E \bowtie D \bowtie P$ as $(E \bowtie D_1 \bowtie P) \cup (E \bowtie D_2 \bowtie P)$. The key idea is that each constituent query can be optimized independently and, due to the correlation of the join attributes, the generated physical plans can be evaluated more efficiently than a single monolithic plan for the whole query. One option, for instance, would be to use different join orders for each plan as follows: $(E \bowtie D_1) \bowtie P$ and $(P \bowtie D_2) \bowtie E$. It is clear that these different join orders are likely to generate less intermediate result tuples than a single plan, which might have a significant impact on query response time.

The previous example illustrates the crux of *selectivity-based partitioning*, the novel query optimization technique that we introduce in this paper. Contrary to the conventional monolithic approach, our proposed technique employs a novel paradigm of divide-and-union for the effective optimization of relational queries: it carefully partitions the base data by examining join selectivities at a finer level of granularity, namely among *relation fragments*, and rewrites an input query to a union of constituent queries that can be optimized separately and more effectively. Overall, this results in a partition-based plan that may improve query response time significantly as it takes into account more detailed information on the join dependencies among relations. We define formally the partitioning problem and present an analysis on the properties of optimal solutions for a simple, yet intuitive cost model that provides useful insights on the characteristics of good partitioning schemes. Based on this model, we develop effective partitioning algorithms that enable an initial solution to the integration of selectivity-based partitioning within an existing query optimizer. More concretely, the key contributions of our work can be summarized as follows:

• **Selectivity-based partitioning.** We introduce selectivity-based partitioning and provide a formal definition of the related optimization problem. Based on our formulation, we present an analysis on the space of possible solutions and derive properties for the optimal partitioning of a given query.

• **Effective Partitioning Algorithm.** Using the results of our analysis, we describe an efficient algorithm for exploring the space of possible solutions and identifying an effective partitioning. Our algorithm is based on the concept of *Iterative Partitioning*, a method that efficiently computes an effective partition while considering only a limited fraction of the total search space.

• **Experimental evaluation of the effectiveness of selectivity-based partitioning.** We present the results of a preliminary experimental study that evaluates the effectiveness of our proposed techniques on real-life and synthetic data sets. The results indicate that selectivity-based partitioning can offer significant improvements compared to a monolithic evaluation plan, while being affordable in terms of optimization time.

To the best of our knowledge, this is the first work that proposes a horizontal partitioning scheme based on the join selectivities among relation fragments.

2. BACKGROUND

Query Model. In this paper, we focus on *chain* queries that join relations $R_0, R_1, \ldots, R_n$ with equality predicates of the form $R_0.A_1 = R_1.A_1\ AND\ R_1.A_2 = R_2.A_2\ AND \ldots R_{n-1}.A_n = R_n.A_n$. That is, every relation R_k (except R_0 and R_n) participates in two joins, with attributes A_k and A_{k+1}. (When no confusion arises, we will use the shorthand notation $R_0 \bowtie R_1 \bowtie \ldots \bowtie R_n$ for the clause of equi-join predicates.) In our model we also consider arbitrary selection predicates on any subset of the attributes of the joined relations. This class of chain queries occurs frequently in real-world applications, and hence presents an interesting case for the development of our optimization techniques. We stress that it is possible to extend our proposed framework to the more general case of *tree-join* queries, where the join-graph forms essentially a tree. The required mathematical machinery gets hairier (e.g., tensors are required instead of matrices), but the essence of our techniques remains the same.

We assume that the values of each attribute $A_k, 1 \le k \le n$, are drawn from a domain $\mathcal{D}_k = \{v_{kj} \mid 1 \le j \le \mathcal{M}_k\}$. We note that we do not make use of any ordering within the domain, i.e., $i \le j$ does not imply $v_{ki} \le v_{kj}$. A $r \times c$ matrix $\underline{F} = (f_{ij})$ is a table of r rows and c columns, where each element f_{ij} is a real number. Given a relation R_k, we define its *frequency matrix* as a $\mathcal{M}_k \times \mathcal{M}_{k+1}$ matrix $\underline{R}_k$, where each element f_{ij} is the number of tuples in relation R_k with $A_k = v_{ki}$ and $A_{k+1} = v_{(k+1)j}$, $1 \le i \le \mathcal{M}_k$, $1 \le j \le \mathcal{M}_{k+1}$. We note that the matrices for R_0 and R_n, which participate in the query with a single attribute, are essentially vectors of dimensions $1 \times \mathcal{M}_1$ and $\mathcal{M}_n \times 1$ respectively. It is a well-known result [15] that the product $\underline{R}_k \cdot \underline{R}_{k+1}$ yields the frequency matrix of the joined tuples $R_k \bowtie R_{k+1}$ on attributes A_k and A_{k+2}, while $\underline{R}_0 \cdot \underline{R}_1 \cdot \ldots \underline{R}_n$ computes the number of tuples in the result of the query.

EXAMPLE 2.1: *Consider a query $Q = R_0 R_1 R_2$, where $\mathcal{M}_j = 2$ for all domains (i.e., each join attribute takes only two values). Assume that the frequency matrices are as follows:*

$$\underline{R}_0 = \begin{pmatrix} 10 & 20 \end{pmatrix} \quad \underline{R}_1 = \begin{pmatrix} 10 & 20 \\ 20 & 40 \end{pmatrix} \quad \underline{R}_2 = \begin{pmatrix} 10 \\ 50 \end{pmatrix}$$

The result size of query Q can be computed as $|Q| = |\underline{R}_0 \cdot \underline{R}_1 \cdot \underline{R}_2| = 55000$. ∎

We note that it is straightforward to incorporate selection predicates in this model, as a predicate on relation R_k essentially scales down the entries of the frequency matrix $\underline{R}_k$. To simplify our presentation, we will assume that all selection predicates have been incorporated in the corresponding frequency matrices and hence the product $\underline{R}_0 \cdot \underline{R}_1 \cdot \ldots \underline{R}_n$ correctly computes the result cardinality of a query.

We note that a real system stores only an approximation of $\underline{R}_j$ in the form of a summary H_j (typically, a histogram) and hence all computations on frequency matrices are carried out using techniques for approximate query processing [2, 14]. For simplicity, we present our techniques assuming that the query optimizer has complete knowledge of the true frequency matrices; an extension to the more general case of summarized distributions is straightforward and omitted in the interest of space.

Query Evaluation Model. We restrict our attention to left-deep evaluation plans as they have become the norm in relational database systems. We represent a left-deep plan as an ordering $O = R_{k_0} R_{k_2} \ldots R_{k_n}$, where R_{k_j} is the inner relation that joins with the result of $R_{k_1} \bowtie R_{k_2} \bowtie \cdots \bowtie R_{k_{j-1}}$. As an example, the ordering $R_1 R_0 R_2$ represents the left-deep plan $(R_1 \bowtie R_0) \bowtie R_2$. Following common practice, we only consider orderings that do not generate cross-products, i.e., every prefix of an ordering is a chain query. An immediate result of this property is that if R_{k_j} is the last relation of the ordering, then $R_{k_j - 1}$ or $R_{k_j + 1}$ must appear in the prefix. We denote the set of all such orderings as $\mathcal{O}(Q)$ and use $cost(O)$ for the evaluation cost of plan O under a suitable cost model. The cost of a query Q can be naturally defined as the minimum over all possible plans, i.e., $cost(Q) = \min_{O \in \mathcal{O}(Q)} (cost(O))$.

We extend the definition of frequency matrices to the case of evaluation plans as follows. The frequency matrix $\underline{O}$ of the ordering $O = R_{k_0}R_{k_2}\ldots R_{k_l}$ ($l \le n$) is defined as the product $\underline{R}_{k'_0}\underline{R}_{k'_2}\ldots\underline{R}_{k'_l}$, where $(k'_0, k'_1, \ldots, k'_l)$ is the sorted order of the indices $\{k_0, k_1, \ldots, k_l\}$. Clearly, $\underline{O}$ is a $\mathcal{M}_{k'_1} \times \mathcal{M}_{k'_l}$ matrix that records the distribution of joined tuples along attributes $A_{k'_1}, A_{k'_l}$.

Given a frequency matrix $\underline{F} = (f_{ij})$, we use $|\underline{F}| = \sum f_{ij}$ to denote the number of tuples that exist in the corresponding relation. If we consider a specific join plan $O = R_{k_0}R_{k_1}\ldots R_{k_n}$, then the total number of intermediate results that are produced during the evaluation of O can be computed as $\|O\| = \sum_{O' \prec O} |\underline{O}'|$, where $\prec$ denotes the prefix relation. (We note that the above expression does not include the count of result tuples $|Q|$.) Similar to the $cost$ function, we define the number of intermediate results for a query Q as the minimum number of intermediate results over all possible plans, i.e., $\|Q\| = \min_{O \in \mathcal{O}(Q)}(\|O\|)$.

EXAMPLE 2.2.: *Consider the query Q of Example 2.1. Under our evaluation model, there are two possible plans , namely, $O_1 = R_0R_1R_2$ and $O_2 = R_2R_1R_0$. (O_1 is equivalent to $R_1R_0R_2$ since both plans generate the same number of intermediate tuples; the same holds for O_2 and $R_1R_2R_0$.) The number of intermediate results in each case is computed as follows:*

$$\|O_1\| = |R_0R_1| = |\underline{R}_0 \cdot \underline{R}_1|$$
$$\|O_2\| = |R_2R_1| = |\underline{R}_1 \cdot \underline{R}_2|$$

■

Partitioning Model. A *partitioning* of relation $R_k \in Q$ is defined as the set $P_{k,c} = \{R_{k1}, \ldots, R_{kc}\}$, where R_{ki} are disjoint subsets of R_k and $\bigcup R_{ki} = R_k$. We will refer to R_k as the *pivot* relation of partitioning $P_{k,c}$. (When no confusion arises, we will drop k, c and use P to denote a partition.) If $Q[R_{ki}]$ is the query that results from Q by substituting R_k with partition R_{ki}, then the following holds:

$$Q \equiv \bigcup_{1 \le j \le c} Q[R_{kj}]$$

We note that the identity holds in multi-set semantics as well, since the partitions are disjoint. A partitioning, therefore, defines a set of queries $Q[R_{ki}]$ that produce the answers to Q without requiring duplicate elimination. Assuming that the constituent queries are evaluated independently, i.e., no computation is shared, we will denote the total number of intermediate results across all queries as follows:

$$\|P_{k,c}\| = \sum_{1 \le j \le c} \|Q[R_{ki}]\|$$

By definition, $\|Q[R_{ki}]\|$ corresponds to an ordering of $Q[R_{ki}]$ that minimizes the number of intermediate results. Since each $Q[R_{ki}]$ is evaluated independently, our model essentially allows different join orders to be used for the constituent queries.

3. SELECTIVITY-BASED PARTITIONING

3.1 Problem Definition

The key idea behind selectivity-based partitioning is to re-write the input query as a union of queries that can be optimized more effectively and hence evaluated more efficiently. In particular, given a query Q, we seek a relation R_k and a partitioning $P_{k,c} = \{R_{ki}\}$, such that the combined evaluation of the constituent queries $Q[R_{ki}]$ is more efficient than a single monolithic plan. The goal, of course, is to "match" the distributions among the partitions and the remaining relations so that the constituent queries can be optimized more effectively than the single monolithic plan. More formally, we define the following optimization problem:

[Query Partition] Consider a query $Q = \{R_1, \ldots, R_n\}$ and let $N > 0$ be a positive integer. We wish to find a pivot relation R_k, a partition budget $c \le N$, and a partitioning $P* = \{R_{k1}, \ldots, R_{kc}\}$, so that the total execution cost of the resulting queries is minimized.

The previous formulation defines the partitioning problem relative to a *specific* query Q. We consider this as a necessary step in order to gain insights on selectivity-based partitioning and apply our techniques in the more general problem of partitioning for an expected workload of queries, or workload-independent partitioning. A second important point is that our formulation involves the partitioning of a *single* relation. Conceptually, it would be possible to partition multiple relations and then define the constituent queries on combinations of partitions. As our analysis shows, however, the problem is not trivial even for the simplified variant of a single partitioned relation. The generalization to partitions over multiple relations is an interesting direction for future work.

Computing a solution to the partitioning problem obviously requires knowledge of the cost model for query execution. The details of a real cost model, however, are not trivial to define and are highly dependent on system configuration. In addition, it is not clear if a solution for a particular cost model will yield general intuitions on the partitioning problem, that will enable its application in a different context. In our approach, we opt for a simplified cost function that enables a more general solution, but at the same time maintains an intuitive (and theoretically sound) connection to the cost factors used in real optimizers. More specifically, we define the cost of a query Q to be equal to the number of intermediate result tuples that are computed during the evaluation of its optimal ordering:

$$cost(Q) = COST^{\mathcal{I}}(Q) = \|Q\|$$

Clearly, the proposed function ignores many of the details of query evaluation in a real system, such as the existence of indices or materialized views, or the use of different physical join operators. As an earlier study [5] has shown, however, optimizing a query for the number of intermediate tuples is equivalent to using more detailed cost metrics that take into account the characteristics of different join operators. These results essentially verify the intuition that an effective execution plan minimizes the number of intermediate results, and justify the use of the proposed function as a generic cost model.

Based on our proposed cost function, we can define the cost of a partition P, as follows:

$$COST^{\mathcal{I}}(P) = \sum_{1 \le i \le c} COST^{\mathcal{I}}(Q[R_{ki}]) = \sum_{1 \le i \le c} \|Q[R_{ki}]\|$$

At this point, we can provide a more concrete definition of the partitioning problem that we address in this paper:

[Query Partition] Consider a query $Q = \{R_1, \ldots, R_n\}$ and let $N > 0$ be a positive integer. We wish to find a relation R_k, a partition budget $c \le N$, a partitioning $P^* = \{R_{k1}, \ldots, R_{kc}\}$, and join orders for queries $Q[R_{k_i}]$, so that the total cost of the resulting queries is minimized, i.e.:

$$P* = \underset{\substack{P_{k,c} \\ 1 \le k \le n, c \le N}}{\operatorname{argmin}} \left(\sum_{1 \le i \le c} \|Q[R_{ki}]\| \right)$$

Clearly, the optimization problem involves two inter-related components: computing the partitioning of a relation, and determining the join orders that essentially minimize the total number of intermediate result tuples for the constituent queries. These individual problems, namely, partitioning and query optimization, are known to be computationally hard in general and, even though we do not provide formal results, our analysis indicates that the combined problem is likely to be hard as well.

An important factor in our proposed approach is the model of *processing* partitioned queries. Consider, for instance, query $Q = R_1R_2R_3R_4$ and its partitioned form $Q = (R_1R_{21}R_3R_4) \cup (R_3R_4R_{22}R_1)$. Clearly, the join R_3R_4 is common in both queries and it would be beneficial to evaluate it only once and use the cached result in the individual plans. In a more general context, the specific form of partitioned queries may enable other types of run-time optimizations that can increase the effectiveness of partitioned query processing. We consider this an important direction for future research, but we do not address this issue in this paper. Here, we focus solely on the important first step of *determining* an effective partitioning.

3.2 Partitioning Algorithm

Overview of our approach. At an abstract level, our partitioning algorithm, termed COMPUTEPARTITION, computes a solution by exploring a space of possible join orderings for the constituent queries and determining an effective partition for each ordering. More formally, let LR_kT be an evaluation plan for the input query Q, where L and T are orderings for the relations in $Q - \{R_k\}$. We will call L the *leading* join ordering, and T the *trailing* join ordering. A *configuration* for R_k is a set $LT = \{(L_1, T_1), \ldots, (L_c, T_c)\}$, where c is the partition count of LT and each (L_i, T_i) defines a valid pair of leading and trailing join orderings. Intuitively, a configuration represents the join plans for the constituent queries if R_k were partitioned in c subsets, i.e., the plan for query $Q[R_{ki}]$ would be $L_iR_{ki}T_i$, $1 \le i \le c$. Given a partitioning P, we will use $COST^{\mathcal{T}}(P, LT)$ to denote the cost of evaluating the constituent queries given the join plans specified by LT. Using the definition of our $COST^{\mathcal{T}}$ metric, this can be written as follows:

$$COST^{\mathcal{T}}(P, LT) = \sum_{1 \le i \le c} \|L_iR_{ki}T_i\|$$

Our proposed partitioning algorithm is based on the concept of an *Optimal Split*, which is defined as the optimal partitioning of a pivot relation for a specific LT. As we discuss later, COMPUTEPARTITION explores a space of different configurations and computes optimal splits as the candidate solutions. To make this process more efficient, our algorithm uses an effective heuristic, termed *Iterative Partitioning*, that limits significantly the number of configurations that need to be examined. In what follows, we first define the optimal split for a given configuration, and then discuss the details of Iterative Partitioning and COMPUTEPARTITION.

Optimal Split. The *optimal split* for a given configuration LT, denoted as $OptSplit(R_k, LT)$, is simply defined as the partition of R_k that minimizes the cost metric for the specific join orderings defined in LT:

$$OptSplit(R_k, LT) = \underset{P=\{R_{k1},\ldots,R_{kc}\}}{\text{argmin}} \left(\sum_{1 \le i \le c} \|L_iR_{ki}T_i\| \right)$$

Intuitively, an optimal split is the optimal solution to the partitioning problem when two of the free parameters are fixed, namely, the number of partitions and the join orderings of the constituent queries.

We now describe a method for computing the optimal split of a specific configuration. The key idea of our technique is that tuples in R_k have *independent* contributions to the total number of intermediate results and hence can be distributed to different partitions using a localized partitioning criterion. We illustrate this idea with a simple example and then state the general formal results.

EXAMPLE 3.1.: *Consider the query $Q = R_1R_2R_3R_4$ and the configuration $LT = (R_1, R_3R_4), (R_3, R_1R_4)$ with respect to the pivot relation R_2. We will assume that $\mathcal{M}_1 = \mathcal{M}_2 = 2$, i.e., the frequency matrix of R_2 looks as follows:*

$$\underline{R}_2 = \begin{pmatrix} X & Y \\ Z & W \end{pmatrix}$$

The partitioning of R_2 can be represented in terms of the frequency matrices of the two partitions:

$$\underline{R}_{21} = \begin{pmatrix} x & y \\ z & w \end{pmatrix} \quad \underline{R}_{22} = \begin{pmatrix} X-x & Y-y \\ Z-z & W-w \end{pmatrix}$$

The number of intermediate result tuples for the first sub-query $R_1R_{21}R_3R_4$ can be computed as follows:

$$\|R_1R_{21}R_3R_4\| = \left|\underline{R}_1 \cdot \begin{pmatrix} x & y \\ z & w \end{pmatrix}\right| + \left|\underline{R}_1 \cdot \begin{pmatrix} x & y \\ z & w \end{pmatrix} \cdot \underline{R}_3\right|$$

Using algebraic manipulations, this can be rewritten as follows:

$$\begin{aligned} \|R_1R_{21}R_3R_4\| &= x \cdot \left(\left|\underline{R}_1 \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}\right| + \left|\underline{R}_1 \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \underline{R}_3\right| \right) \\ &+ y \cdot \left(\left|\underline{R}_1 \cdot \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}\right| + \left|\underline{R}_1 \cdot \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \cdot \underline{R}_3\right| \right) \\ &+ z \cdot \left(\left|\underline{R}_1 \cdot \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}\right| + \left|\underline{R}_1 \cdot \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \cdot \underline{R}_3\right| \right) \\ &+ w \cdot \left(\left|\underline{R}_1 \cdot \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right| + \left|\underline{R}_1 \cdot \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \underline{R}_3\right| \right) \\ &= x \cdot X_1 + y \cdot Y_1 + z \cdot Z_1 + w \cdot W_1 \end{aligned}$$

As seen by the expression, the block of x tuples from R_2 generates a number of intermediate results that is independent of the contributions of other R_2 tuples. In particular, the factor X_1 depends solely on the leading and trailing join sets, and on the position of the x tuples in the frequency matrix $\underline{R}_2$. A similar expression can be derived for the intermediate results of the second partition:

$$\begin{aligned} \|R_3R_{22}R_1R_4\| = (X-x) \cdot X_2 + (Y-y) \cdot Y_2 + \\ (Z-z) \cdot Z_2 + (W-w) \cdot W_2 \end{aligned}$$

The total number of intermediate results can be reduced, therefore, by assigning all X tuples to the partition with the minimum X_i weight, which in turn can be computed from the configuration only and independently of the assignments of other tuples. A similar observation holds for the Y, Z, W tuples of R_2. ■

More formally, we can state the following result for a general configuration LT:

LEMMA 3.1. *Consider a query plan LR_kT and let f_{ml} denote the number of tuples in R_k with $A_k = v_{km}$ and $A_{k+1} = v_{(k+1)l}$. The total number of intermediate result tuples for the query plan LR_kT is computed as:*

$$\|LR_kT\| = \|L\| + \sum_{\substack{1 \le m \le \mathcal{M}_k \\ 1 \le l \le \mathcal{M}_{k+1}}} f_{ml} \cdot g(L, m, l, T) \tag{1}$$

where $g(L, m, l, T)$ is an expression involving the frequency matrix of L, the frequency matrices of relations in T, and a matrix whose contents depend solely on m, l.

Overall, Lemma 3.1 asserts the following important property: tuples of R_k that cover different parts of the value domain have independent contributions to the total number of intermediate results. To minimize the total number of intermediate results, therefore, it suffices to assign all tuples that cover $(v_{km}, v_{(k+1)l})$ to the partition with the minimum contribution for this particular region of the value domain. This is captured formally in the following theorem:

THEOREM 3.1. *Consider a configuration $LT = \{(L_i, T_i), 1 \le i \le c\}$ for relation R_k. Let $P^* = \{R_{k1}, \ldots, R_{kc}\}$ be a partitioning where all f_{ml} tuples of R_k that cover $(v_{km}, v_{(k+1)l})$ are assigned to partition R_{ki^*}, where $i^* = \text{argmin}_{1 \le i \le c}(\|L_i\| + f_{ml} \cdot g(L_i, m, l, T_i))$. Then P^* is an optimal split w.r.t. configuration LT, i.e., $P^* = OptSplit(R_k, LT)$.*

Theorem 3.1 is key in our framework since it provides the method for computing the optimal split for any configuration LT. In addition, as the following result shows, it has an important ramification regarding the number of configurations that need to be considered in order to compute an optimal partitioning.

LEMMA 3.2. *Given an ordering L, let L^* be the optimal, in terms of the $COST^{\mathcal{T}}$ function, evaluation plan for computing the same join as L. Let $LT = \{(L_i, T_i), 1 \le i \le c\}$ be a configuration and define $LT^* = \{(L_i^*, T_i), 1 \le i \le c\}$ as the configuration with the optimal leading join orderings and the same trailing join orderings as LT. The following holds for the costs of the optimal splits P and P^* for LT and LT^* respectively: $COST^{\mathcal{T}}(P, LT) \ge COST^{\mathcal{T}}(P^*, LT^*)$*

Lemma 3.2 simply states that it is always possible to get a more effective partitioning by considering the optimal orderings for the leading join sets. (This property can be seen as an extension of the local optimality principle that is used by query optimization techniques based on dynamic programming [19].) In essence, this result limits considerably the number of configurations that can lead to effective solutions, as it is never advantageous to examine a configuration that does not contain the optimal plans for the leading join orderings. For the remainder of the paper, therefore, we will consider only configurations that contain the optimal leading join orders for every partition. (These optimal orderings can be computed efficiently using well known optimization techniques [12, 19].) A natural question to ask is whether the same property holds for the trailing join orders, i.e., if the optimal partitioning is achieved for configurations with optimal trailing join orders. Unfortunately, a simple counter-example can demonstrate that this is not the case; intuitively, each partition R_{ki} causes a set of frequencies to become zero in the output of $L_i R_{ki}$, which may lead to an optimal partitioning that does not contain the optimal ordering for T_i.

Up to this point, we have been vague on the form and complexity of expression $g(\cdot)$ that is used in Theorem 3.1. At an abstract level, $g(\cdot)$ involves the computation of t frequency matrix products, where t is the number of tables in the trailing join order, and product i is formed from product $i-1$ with the addition of a single frequency matrix. (This is illustrated in Example 3.1 in the definition of X_1.) We expect these operations to be very efficient in a real system, as the products are computed on highly-compressed summarized distributions using techniques for approximate query answering. Moreover, as we discuss later, our partitioning algorithm can control effectively the overhead of query optimization by introducing an early-stopping condition based on time deadlines. We revisit these points in Section 4.2, where we evaluate experimentally the overhead of our optimization techniques.

Partitioning Algorithm. Our previous discussion focused on a constrained variant of the partitioning problem where the pivot relation and the join orderings are fixed. In the remainder of this section, we leverage our analysis and introduce a partitioning algorithm, termed COMPUTEPARTITION, for the general Query Partition problem. Clearly, the general version of the problem suffers from a combinatorial explosion of the search space, as an exhaustive algorithm will have to consider all possible pivot relations and all possible leading and trailing join orderings. To address this issue, our partitioning algorithm employs an effective heuristic, termed *Iterative Partitioning*, that essentially computes high-quality solutions based on the optimized orderings of the leading join sets and a limited subset of possible trailing join orders.

```
Procedure ITERPART($S$)
Input: State $S = (c, k, L)$
Output: Cost of a partition that is derived by
    Iterative Partitioning on $S$.
begin
1.  Compute an initial partitioning $P$
2.  $cost^*$ := $\infty$; $iter$ := 0
3.  do
4.      $iter$ := $iter + 1$;
5.      Let $(L_i \bowtie R_{ki})T_i^*$ be the optimal plan for subset $R_{ki}$
6.      $LT^*$ := $\{(L_i, T_i^*), 1 \le i \le c$
7.      $P^*$ := OPTSPLIT($LT^*$, $R_k$) // Compute optimal split
8.      $cost^*$ := $COST^{\mathcal{T}}(LT^*, P^*)$
9.  until ( $P^*$ is unmodified OR $iter = iterlim$ )
10. return $cost^*$
end
```
Figure 2: Iterative Partitioning

We now proceed to describe our approach in detail, starting from our Iterative Partitioning heuristic. Consider a possible configuration $LT = \{(L_i, T_i)\}$ and let $P = \{R_{ki}, 1 \le i \le c\}$ be the optimal split for LT. Assume that T_i^* denotes an ordering for the relations in T_i, such that $(L_i \bowtie R_{ki})T_i^*$ is an optimal query plan, i.e., it minimizes the number of intermediate join tuples. It is straightforward to show that if $LT' = \{(L_i, T_i^*)\}$ is the configuration with the same leading join plans as L and the optimized orderings T_i^*, then $COST^{\mathcal{T}}(P, LT) \ge COST^{\mathcal{T}}(P, LT^*)$. In other words, it is possible to derive a more efficient solution by *solely* re-ordering the trailing join plans according to each prefix result $L_i \bowtie R_{ki}$ (without modifying the split of the pivot relation R_k). This re-ordering, however, essentially modifies the configuration and hence raises the opportunity for computing a new, more effective split. More formally, this can be stated as follows:

LEMMA 3.3. *Let LT and P be defined as previously, and let LT^* be the modified configuration with the optimized trailing join plans. If $P^* = OptSplit(LT^*, R_k)$ is the optimal split for the new configuration, then $COST^{\mathcal{T}}(P, LT) \ge COST^{\mathcal{T}}(P^*, LT^*)$.*

Lemma 3.3 forms the basis of our Iterative Partitioning heuristic. The key idea is to create an initial partitioning, e.g.,based on a random ordering of the trailing join plans, and subsequently improve on it by successively re-optimizing the trailing plans and re-partitioning the pivot relation. This is illustrated in Figure 2 that depicts the pseudo-code for Iterative Partitioning. The algorithm applies Lemma 3.3 repeatedly on the current partitioning, and terminates when the cost of the computed solution is not modified between iterations, or when a pre-specified limit of iterations is reached. It is interesting to note that Iterative Partitioning presents an attractive option for integrating selectivity-based partitioning with the cost model of a real optimizer. In particular, it is possible to employ a realistic cost model for optimizing the leading join orders and re-optimizing the trailing join orders in each iteration, and to rely on the simplified function $COST^{\mathcal{T}}$ and Theorem 3.1 only for re-routing the tuples among different partitions. This approach opens up an interesting direction for future work, as it is clearly a feasible approach for applying selectivity-based partitioning in an existing optimizer.

```
Procedure ComputePartition($Q$,$N$)
begin
1.$cost^* = COST^{\mathcal{T}}(Q)$ // Cost of a monolithic plan
2. init_states($\mathcal{S}$)
3.while $\mathcal{S} \neq \emptyset$ do
4.   $S$ := $\mathcal{S}.pop()$ // State with least $score_{\mathcal{C}}$ and least $c$
5.   if ($score_{\mathcal{C}}(S) < cost^*$) then
6.      $cost^*$ := $min(cost^*$, IterPart($S$))
7.   end-if
8.   for $1 \leq i \leq c, R \in Q - (L_i \cup \{R_k\})$ do
9.      $\mathcal{S} \leftarrow t_I(S, L_i, R)$;
10.  done
11.done
end
```

Figure 3: Algorithm ComputePartition**.**

Our proposed partitioning algorithm, termed ComputePartition, uses Iterative Partitioning as a method of exploring efficiently the space of possible partitions. The pseudo-code for ComputePartition is shown in Figure 3. A state S in the search space is encoded as a triplet $S = (R_k, c, L)$, where R_k is the candidate pivot relation, c is the number of partitions, and L is a set of c leading join sets for R_k. (Again, the algorithm only considers the optimized orderings for the leading join sets.) At an abstract level, the algorithm considers each triplet S in the set of open states $\mathcal{S}$, derives an effective partitioning using the Iterative Partitioning method on S, and records the best solution in variable $cost^*$. $\mathcal{S}$ is initially populated with all triplets $(R_k, c, L_s(c, k))$, for all $1 \leq k \leq n$, $1 \leq c \leq N$, and $L_s(k, c) = \{(R_{i_1}), (R_{i_2}), \ldots, (R_{i_c}) | R_{i_j} \in Q - \{R_k\}\}$ (function init_states). To make the search more efficient, $\mathcal{S}$ is maintained as a priority queue based on the *completion score* of each state which is defined as $score_{\mathcal{C}}(S) = \sum_{1 \leq i \leq c} \|L_i\|$, i.e., the sum of intermediate result tuples for all the (optimized) leading join orders. It is straightforward to show that $COST^{\mathcal{T}}(P) \geq score_{\mathcal{C}}(S)$ for any partition P that is derived based on the leading join sets in S. The score metric, therefore, serves as a heuristic for guiding the search process toward states that are likely to yield effective solutions. Furthermore, the algorithm does not apply Iterative Partitioning on S if $score_{\mathcal{C}}(S) \geq cost^*$ (line 5), as it provably cannot yield a more effective solution.

To explore the state space, ComputePartition uses a transition function t_I that essentially inserts one more relation in exactly one of the input leading sets (lines 8-10). More formally, consider a state $S = (R_k, c, L), L = \{L_1, \ldots, L_c\}$ and a specific join ordering $L_j \in L$ $(1 \leq j \leq c)$. If $R \neq R_k$ is a relation that does not yet appear in L_j, then the result of $t_I(S, L_j, R)$ is a new state $S' = (R_k, c, L')$ where L' contains exactly the same leading join sets as L except for $L'_j = L_j \cup \{R\}$. This transition function enables the efficient computation of the tuple counts and frequency matrices for the leading orders of each state S, as $\underline{L'_j}$ can be computed with a single matrix multiplication between $\underline{L_j}$ and $\underline{R}$.

It is interesting to note that ComputePartition operates in a progressive mode: it computes an initial partitioning (which corresponds to $c = 1$, i.e., the optimal monolithic plan) and subsequently discovers better solutions as more of the search space is explored. This enables the option of calling the algorithm with a time-based deadline, where the search is terminated after a pre-specified time interval has expired and the best current solution is returned. As our empirical study shows, this is a simple, yet effective solution for controlling the overhead of optimization while retaining the benefits of partition-based processing.

4. EXPERIMENTAL STUDY

In this section, we present the results of an experimental study that we have conducted on real-life and synthetic data for evaluating the effectiveness of our partitioning technique.

4.1 Methodology

Techniques. We have completed a prototype implementation of the ComputePartition algorithm that we introduce in this paper. In all the experiments that we present, ComputePartition uses a maximum of four subsets ($N = 4$) for computing a partitioning, and performs exactly one iteration for Iterative Partitioning. For comparison purposes, we have also implemented an exhaustive search algorithm, termed OptimalPartition, that explores all possible solutions and identifies an optimal partitioning for the input query.

Data Sets. We have based our experimental study on synthetic and real-life data sets.

Synthetic Data. We have generated several synthetic data sets of varying characteristics. A data set consists of 60 relations in total, each relation having 10^4, 10^5, or 10^6 tuples. Since our intent is to study the performance of our partitioning technique, we exclude the effect of statistics errors and assume that the frequency matrix of each relation can be accurately represented by an equi-width 2-dimensional histogram of 100 buckets. The bucket frequencies in each histogram are assigned according to a zipfian distribution, with a specific skew parameter z that is the same for the whole data set. In our experiments, we have generated four data sets with skew values 0.5, 1.0, 1.5, and 2.0. (Each data set is denoted as $D(z)$, where z is the value for the skew parameter.)

Real-Life Data. We have used a subset of the well known SwissProt database as our real-life data set. More specifically, we have generated a relational schema containing the following tables: *RA* (2,541,749 tuples), *OC* (512,487 tuples), *KW* (385,634 tuples), *DR* (593,933 tuples), and *ID* (50,000 tuples). A tuple in *ID* represents a protein entry in SwissProt, while each of the *RA*, *OC*, *KW*, and *DR* tables essentially record the inclusion of annotations in each entry[1]. Tables *RA*, *OC*, *DR*, and *KW* have a foreign key dependency to table *ID*, with an average of 50, 10, 11, and 7 joining tuples per tuple in *ID*.

Workloads. For the synthetic data sets, we have generated a workload of 300 random join queries consisting of 3-5 relations each. For SwissProt, we have experimented with various workloads but we only present results for queries that join the entry table *ID* with all possible combinations of three annotation tables. (We have selected this particular workload as it yields an interesting range of performance measurements for our technique.) The average count of intermediate result tuples for the optimized monolithic plans is $7 \cdot 10^{15}$ for the synthetic workload, and $3 \cdot 10^6$ tuples for the SwissProt queries. We have also experimented with workloads that contain random selection predicates on the base tables and our results have been qualitatively the same.

Metric. We quantify the performance of our partitioning scheme by the average reduction in the number of intermediate result tuples over the queries in the workload. More specifically, if LT and P are the configuration and partitioning respectively returned by

[1]Since multiple entries may share the same annotations, these tables essentially record only the key of the entry and the key of the annotation.

COMPUTEPARTITION, then the relative reduction is simply computed as $r(Q) = (COST^{\mathcal{T}}(Q) - COST^{\mathcal{T}}(P, LT))/COST^{\mathcal{T}}(Q)$, where $COST^{\mathcal{T}}$ is the function defined in Section 3. A positive $r(Q)$ represents a percentage of improvement on the cost of Q, while $r(Q) = 0$ implies that the best partitioning P is identical to the monolithic plan, i.e., $c = 1$. Note that our algorithms never return a partitioning that has cost greater than $cost(Q)$, and therefore the $r(Q)$ metric will always lie in the range $[0, 1]$. In our experiments, we report the average reduction over all the queries of a workload.

4.2 Experimental Results

In this section, we present the results of the experimental study that we conducted for evaluating the effectiveness of selectivity-based partitioning. In what follows, we first present a sensitivity analysis of our technique for varying parameters of the synthetic data sets, and then conclude with the results on our real-life data set.

Reduction of intermediate result tuples. Figure 4(a) shows the reduction metric for the OPTIMALPARTITION and COMPUTEPARTITION algorithms as a function of the data skew in our synthetic data. Our results demonstrate that partition-based processing offers significant reductions in the number of intermediate tuples compared to an optimal monolithic plan. For the $D(1.5)$ data set, for instance, the partitioned plan achieves an average reduction of 45% compared compared to an optimal monolithic plan. For certain queries, in fact, we observed improvements of up to 90% — obviously a very significant savings in the count of intermediate join tuples. The results also show that COMPUTEPARTITION yields consistently near-optimal improvements and is out-performed only marginally by OPTIMALPARTITION. This clearly demonstrates the effectiveness of the Iterative Partitioning heuristic in computing low-cost partitions while exploring only part of the search space.

The previous results indicate an increasing trend in the performance of our partitioning schemes as data skew becomes higher. Intuitively, selectivity-based partitioning offers the most benefit when partitions eliminate values with high frequencies in the respective trailing join sets, thus nullifying their contribution to the number of intermediate join tuples. As the skew is lowered, values are distributed more evenly and therefore the benefit decreases since more frequencies contribute significantly to the count of intermediate results. Given that real-life data distributions are characterized by non-uniformity, however, our results demonstrate that selectivity-based partitioning is likely to yield significant performance improvements in practice.

It is interesting to note that the computed solutions always consist of two partitions and, in the vast majority of cases, the join orderings are mirrored, i.e., the solution has the form $(LR_{k1}T) \cup (TR_{k2}L)$. The sub-queries, therefore, can be evaluated very efficiently by materializing the common sub-expressions L and T and sharing their computation between the two plans. As we mentioned in Section 3, this is one possible optimization for the integration of selectivity-based partitioning in the architecture of a typical DBMS. There are clearly several practical issues involved in this endeavor, and we plan to investigate them as part of our future work on this problem. In this paper, we have focused on the challenging, and largely orthogonal, first step of *determining* an effective partitioning.

# of Joins	3	4	5	6
Exec Time (seconds)	0.078	0.405	1.977	9.343

Table 1: Execution times for COMPUTEPARTITION

Optimization overhead. Up to this point, we have evaluated the effectiveness of selectivity-based partitioning when COMPUTEPARTITION runs to completion and thus explores a significant fraction of the solution space. In this case, however, the overhead of the partitioning algorithm may be prohibitive for the stringent time constraints of an optimizer. This is clearly shown in Table 1 that lists the average execution time of COMPUTEPARTITION for queries of different complexity on data set $D(1.5)$. The results verify the exponential growth in the number of explored states and demonstrate that a complete execution of COMPUTEPARTITION may not be a feasible option in practice.

As we have discussed in Section 3.2, our partitioning algorithm can also operate in a "deadline" mode where it returns the best solution that has been found within a given time limit. To evaluate this approach, we have measured the effectiveness of the computed partitioning as we vary the deadline given to COMPUTEPARTITION. We have based our evaluation on synthetic data and we have used two workloads: Q_L, which is the main synthetic workload, and Q_H, an additional set of 300 queries that contain 6-8 relations each. The latter represents a "difficult" workload with an increased search space and is included in order to evaluate the sensitivity of the deadline approach to the number of possible solutions.

Figure 5 shows the average reduction metric of the generated partitions as a function of the deadline, for the synthetic workloads Q_L and Q_H on data set $D(1.5)$. (The results for the other data sets are similar.) Clearly, COMPUTEPARTITION can generate effective solutions even under stringent constraints on the overhead of optimization. Given a moderate deadline of 1 second, for instance, the computed partitioned plans result in an average reduction of 45% for queries with 3-5 relations, and 18% for queries with 6-8 relations. As expected, the same time deadline yields better results for queries with fewer relations as COMPUTEPARTITION manages to explore a larger portion of the smaller search space within the limited time budget. Still, our results validate the effectiveness of this "progressive" mode of optimization and demonstrate the feasibility of selectivity-based partitioning as a practical optimization technique.

Performance evaluation on real-life data. We now present a limited set of experiments on our subset of the SwissProt database. As we have discussed in Section 4.1, our test workload contains the 4 join queries, denoted as $Q_0, \ldots, Q_3$, between the *ID* table and three of the annotation tables. We note that we experimented with workloads that contained random selection predicates as well, but we observed similar results and we omit them from our presentation in the interest of space.

Figure 6 shows the reduction metric of the computed partitioning and the actual savings in intermediate result tuples for the four queries in our test workload. (In this experiment, COMPUTEPARTITION executed to completion with an average optimization time of 120msec.) Overall, the results demonstrate that selectivity-based partitioning can be effective in reducing the count of intermediate result tuples in real-life data sets. For the test query $Q_3 : ID \bowtie OC \bowtie RA \bowtie KW$, for instance, the partitioned plan generates 750,000 less intermediate tuples than the monolithic case, resulting in 24% of savings. As our results indicate, however, not all queries are amenable to this type of partition-based optimization. An example is query Q_2: $ID \bowtie OC \bowtie KW \bowtie DR$, where selectivity-based partitioning enables a significantly lower reduction of 10%. Still, depending on the width of the generated tuples, this lower reduction can result in significant savings in storage space and hence improve the performance of query processing.

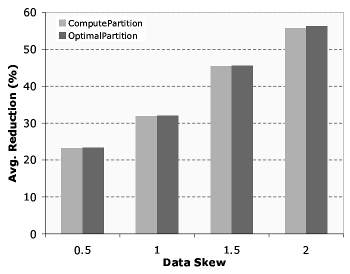

Figure 4: Performance of COMPUTEPARTITION varying skew

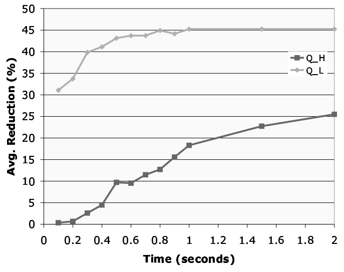

Figure 5: Performance of partitioning for varying time budget

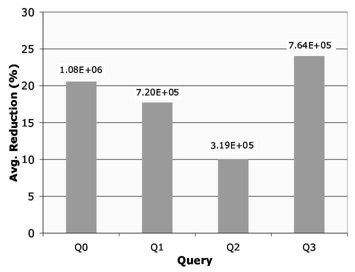

Figure 6: Performance of COMPUTEPARTITION on real-life data.

5. RELATED WORK

The optimization of relational queries has been a topic of active research and previous studies have proposed a host of effective techniques [3, 9, 10, 13, 17, 18, 19, 20] that address different aspects of the general problem. The proposed techniques, however, operate within the physical database schema and the generated plans refer to the already existing data structures (e.g., relations, indices, or materialized views). Our approach, on the other hand, is based on the partitioning of base tables, according to the join selectivities between groups of tuples, and the rewriting of the query as a union of queries over the partitions. As our results have shown, this approach can lead to significant savings in the number of intermediate result tuples.

Our proposed techniques resemble the concept of *horizontal table partitioning* that has been applied successfully in the context of parallel and distributed databases [7, 8, 11]. In a nutshell, relations are partitioned across the different nodes of a parallel system and queries are rewritten accordingly to use the fragments of each node. The goal, of course, is to spread the load of a query across the system and thus increase the effectiveness of parallel execution. The key difference from our technique, however, is that the routing of tuples depends solely on the contents of the partitioned relation and hence does not take into account the correlations among different relation fragments. Dobra et al. [4] have explored the horizontal partitioning of relational data in the context of streaming databases. Their technique, however, is based on a different objective function than the one we consider in this paper: minimizing the sum of products of self-join sizes across all partitions. Hence, their cost model does not take into account the ordering of relations in the query plan, which forms a key factor of our optimization problem.

Recent studies on adaptive query processing [1, 6] have looked at the related problem of dynamically modifying a query plan in order to reduce query response time. The proposed techniques target mainly queries over streaming data sources, where distribution information is often not available, and determine the join order of upcoming tuples based on the join selectivities of already received tuples. As a result, the order of tuple arrival can affect the join plans that are chosen by the adaptive optimizer. Our approach, on the other hand, focuses on the case where distribution statistics are available and determines a static value-based routing of tuples based solely on their frequency distribution.

6. CONCLUSIONS

In this paper we initiate the study of selectivity-based partitioning, a novel approach to query optimization that adopts a divide-and-union approach to query evaluation. We define formally the optimization problem and present an analysis on the characteristics of an optimal solution. Based on this analysis, we develop an efficient algorithm for computing an effective partitioning of the input query while considering a limited fraction of the total search space. Our experimental results verify the effectiveness of selectivity-based partitioning and demonstrate its potential as a paradigm for query optimization.

7. ACKNOWLEDGMENTS

The author wishes to thank Shivnath Babu, Pedro Bizarro, and Minos Garofalakis for their helpful comments on earlier drafts of this paper.

8. REFERENCES

[1] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *ACM SIGMOD*, 2004.
[2] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *VLDB*, 2000.
[3] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *VLDB*, 1996.
[4] A.Dobra, M.Garofalakis, J.Gehrke, and R. Rastohi. Processing complex aggregate queries over data streams. In *ACM SIGMOD*, 2002.
[5] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*, 1995.
[6] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.
[7] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, 1986.
[8] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6), 1992.
[9] G. Graefe and D. J. DeWitt. The exodus optimizer generator. In *ACM SIGMOD*, 1987.
[10] A. Halevy. Answering queries using views: A survey. *Intl. Journal on Very Large Data Bases*, 10(4), 2001.
[11] K. A. Hua and C.Lee. An adaptive data placement scheme for parallel database computer systems. In *VLDB*, 1990.
[12] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.
[13] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *ACM SIGMOD*, 1990.
[14] Y. E. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query Answers. In *VLDB*, 1999.
[15] Y. E. Ioannidis. "Universality of Serial Histograms". In *VLDB*, 1993.
[16] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
[17] R. Krishnamurthy, B. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.
[18] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *ACM SIGMOD*, 1992.
[19] P. G. Selinger, M. M. Astrahan, R. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM SIGMOD*, 1979.
[20] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1), 1988.