Histogram Guided Interactive Query Evaluation

Donko Donjerkovic
Raghu Ramakrishnan

Technical Report #1419

August 2000

Technical Report: CS-TR-00-1419. UW-Madison

Histogram Guided Interactive Query Evaluation

Donko Donjerkovic, Raghu Ramakrishnan
Department of Computer Sciences
University of Wisconsin–Madison

Abstract

The data analysis process is interactive. Analysts are more interested in seeing a subset of query answers quickly than in seeing all answers after a long waiting period. To effectively support such interactive queries, we propose new join execution strategies that minimize the time to produce the first N answers. Our techniques deliver early result tuples at a significantly higher rate than traditional pipelined join plans, as demonstrated by experiments. This improvement is a result of better memory utilization, which is in turn made possible by exploiting database statistics to prioritize tuples that are the largest contributors to the join result. To incorporate these new techniques into a cost-based optimizer, we derive cost formulas and suggest an optimization strategy that seeks to minimize the response time for early query answers. We also show that system statistics can be exploited, surprisingly enough, to significantly improve the *total execution time* of traditional blocking operators such as Hybrid Hash join and hash-based group-by.

1 Introduction

It is widely recognized that the information analysis tools should provide quick response times since they are used in an iterative manner. Analysis usually starts with broad queries that are subsequently refined based on the initial answers. Database systems used as backends of such tools must, therefore, be able to provide interactive response times as well.

Data analysts typically examine large databases in search of common trends. For example, one might compare total sales on a monthly basis. Two common methods for detecting large scale trends are: (1) Aggregate operations such as sum, count, and average, and (2) Data visualization. While it is evident how aggregating data reduces information complexity, data visualization does this as well, since large distributions can be presented in a graph where data trends may be evident at a glance. Both of these methods can be made interactive, as explained next.

Online aggregation techniques produce approximate aggregate answers with increasing precision over time [8, 6]. The objective of an online aggregation system is not to minimize the time to produce the final answer, but to minimize the time to achieve a small confidence interval. Because the confidence interval is proportional to $1/\sqrt{n}$, where n is the number of processed tuples, the basic objective of online aggregation is equivalent to the objective of minimizing time to produce a certain number of answers. On the other hand, database visualization systems, such as DEVise [10], can be interactive by displaying tuples as soon

Price	Product Id
15	1
10	3
14	1
9	4
6	2
9	3
14	1
15	1

Figure 1: Example Sales table.

Product Id	Manufacturer
1	Bayer
2	Johnson & Johnson
3	Schering-Plough
4	Merck

Figure 2: Example Products table.

as they are produced. This is, of course, under the assumption that answers to a query are being produced as soon as the query is issued. It follows that, in order to be useful for interactive visualization, database systems must provide an option of optimizing queries for the first few (N) answers. Some commercial systems support such optimization, mostly by choosing pipelined plans with Nested Loops joins, as discussed in the section on related work (Section 6).

As described, optimizing for N answers is needed in online aggregation and data visualization environments. Querying mode in which the emphasis is on producing a subset of answers quickly is called *partial* or *interactive* querying. In most scenarios, optimizing for N answers forces an optimizer to use a pipeline of Nested Loops joins. Unfortunately, Nested Loops joins are inefficient, mostly due to the non-locality of page references. For example, when memory is lacking, every index lookup in the Index Nested Loops (INL) join can result in one I/O operation. Because some form of an INL join is required for fast delivery of the first block of answers, we exploit the available system statistics to significantly improve memory utilization of the INL join.

We now consider the following motivating example.

Example 1.1 Consider a business intelligence query that explores the distribution of sale prices by various drug manufacturers, based on point of sale data. (This example is motivated by real world data that we use to evaluate our technique.) The sales table has *price* and *productId* attributes, among several others. The product table has *productId* (primary key with an index) and *manufacturer* attributes, among other fields. Scaled down examples of sales and product tables are shown in Figures 1 and 2, respectively. Notice that the most common *productId* in the sales table is 1.

Suppose we want to execute the query that asks for $(price, manufacturer)$ pairs:

```
select price, manufacturer
from sales s, products p where s.productId = p.productId
```

The system needs to perform an equi-join on *productId* between Sales and Products tables. A blocking join operator, such as hash join, is not suitable in an interactive environment due to a large initial "think" time. Index Nested Loops join, with Products as the inner table, is a better option since it will immediately start producing answers. However, assume that just two pages of memory are available for the join, one for the inner table and one for the outer table. (We use such small numbers since a more realistic scenario would be harder to

follow.) Suppose the number of tuples per page is two for both relations, and that a user wants to see five answer tuples as soon as possible. We now calculate the total number of I/Os needed to produce five answers using INL join. There are five I/Os on Products table, one for each index lookup needed to match the first five tuples in Sales. In addition, there are three I/Os on Sales, as we need to read three pages sequentially before we reach the fifth tuple (two tuples per page). This brings the total to eight I/Os before the five answer tuples are produced.

To produce the five answer tuples quicker, we propose the following strategy. Identify one page from Products (since that is the memory available) that is most frequently referenced by Sales. This is clearly the first page of Products (with tuples 1 and 2), since it has five references, compared with three references to the second page. Scan the sales table, and at the same time produce answers for tuples that have a match on this page. In this case we have four I/Os on Sales (since the whole outer must be scanned to produce five answers) and one I/O on Products, a total of five I/Os. By better utilizing the available memory we were able to reduce the number of I/Os from eight to five. Savings are more impressive when more realistic numbers are used, as illustrated in the experimental section (Section 3). □

The main contribution of the work described in this paper is to propose and demonstrate effectiveness of techniques that use tuple frequencies for better memory utilization. Tuple frequencies are readily available in current database systems as a part of system statistics, usually in forms of histograms [14].

Histograms and other statistical summaries, such as random samples and wavelets, have been proposed and used in database systems primarily for query optimization. To the best of our knowledge, this is the first time that the use of distribution summaries is shown to significantly improve the performance of query execution.

The rest of the paper is organized as follows: Section 2 describes techniques for performance improvements of partial query evaluation. Experimental evaluation follows in Section 3. We then propose an optimization framework for partial query evaluation in Section 4, which takes into account properties of the new operator evaluation methods. Even though our work was primarily motivated by inefficiencies of partial query evaluation, we found that database statistics can be used to speed up the computation of all query answers also, as discussed in Section 5. Finally, we discuss related work, outline future work and conclude.

2 Statistics Guided Partial Query Evaluation

In this section, we focus on partial query evaluation for which blocking relational operators are not applicable, as explained in the introductory section (1). Our discussion is focused on common foreign key joins, but the results are easily applicable to general equi-joins as well. We first describe how statistical information helps minimize the I/O cost to produce the first N answers in single foreign key joins, and then extend the ideas to address multiple foreign key joins.

2.1 Single Foreign Key Joins

Our discussion in this section is applicable to INL and Simple Nested Loops joins. As illustrated in Example 1.1, Nested Loops joins have poor locality of memory references on

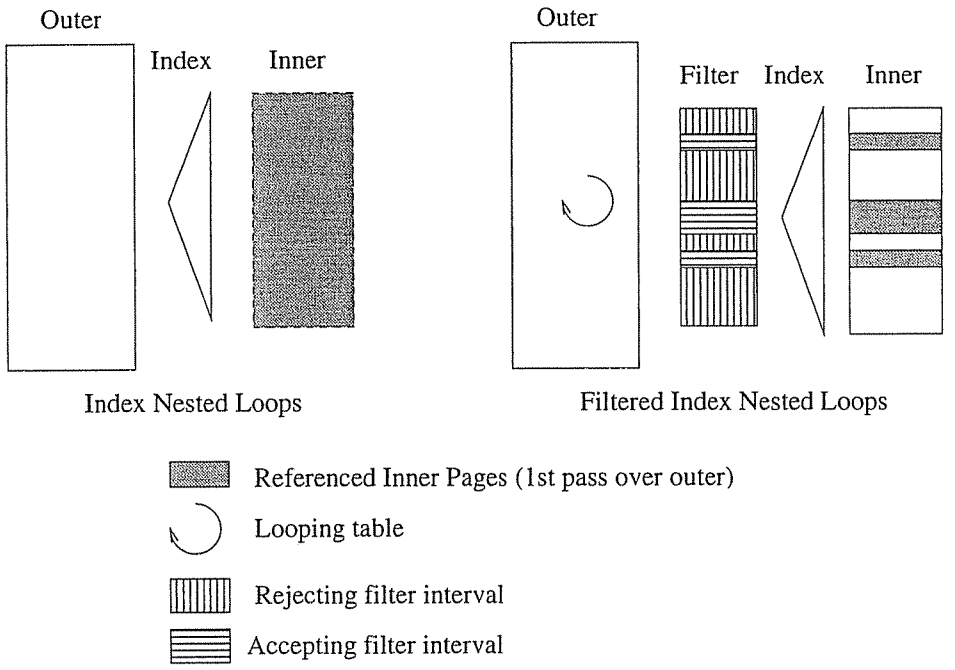

Figure 3: Comparison of traditional and Filtered Index Nested Loops joins.

the inner relation when the available memory is not sufficient. In that example, we illustrated a strategy to improve memory utilization. We now summarize our observations. We propose to use a multi-phase join algorithm. The first *join phase* consists of two operations:

1. We identify the set of pages of the inner relation that has the most references in the outer relation. Tuples that occupy that set of pages can be defined by a relational selection on the join column which we call the *join filter*. (See Section 2.1.1.)

2. We sequentially scan the outer relation; if a tuple satisfies the join filter, an index lookup is performed and an answer is produced, otherwise the tuple is skipped.

If more answer tuples are desired, the next join phase is performed, which differs from the previous only in the join filter; it now includes the set of pages that are most frequently referenced and do not satisfy any previous filter.

We call such a join operation *Filtered Index Nested Loops join*. Note that our algorithm partitions the inner relation into disjoint blocks, and joins each block with the entire outer relation. A conceptual comparison between the traditional INL join and the filtered version is depicted in Figure 3. We note that index lookups of a Filtered INL join are limited to the set of join values satisfying the join filter. As a consequence, the set of referenced pages is limited. Because some outer tuples are rejected, i.e., no index lookup is attempted, additional scans over the outer relation may be required. On the other hand, a traditional INL join requires only one scan of the outer relation, and it repeatedly references all the pages of the inner relation. The major difference between these two algorithms is the *usage of database statistics*. While a traditional INL algorithm, makes *no* use of database statistics, a Filtered INL join uses frequency information to define the memory content in order to maximize the number of early answers. We reiterate that none of the traditional join

operators take advantage of readily available database statistics. Most of these operators could be enhanced by filtering. Perhaps the cleanest example would be the Filtered Block Nested Loops join. This algorithm is identical to the Block Nested Loops join except that a block would always contain the (next) most frequently referenced pages in the outer relation, not just a contiguous set of pages on disk. A disadvantage of Filtered Block Nested Loops join is that it requires a full scan of the outer relation before any answers are produced, which is why we focus on the INL join instead.

The key idea in join filtering is to construct a filter such that it contains the current set of the most frequently referenced pages. We discuss how to do this next.

2.1.1 Constructing a Join Filter for Index Nested Loops Join

As indicated before, a join filter ideally contains a set of the most referenced inner pages that remain to be joined. Therefore, to create an ideal join filter one would need to order pages of the inner relation by the number of references. In a realistic scenario, it would be too expensive to construct an exact join filter, so we consider the following approximations:

1. Maintain (small) histograms [14] to represent approximate reference counts for each page.

2. Use existing histograms on tuple values, together with some information about the tuple layout on pages in order to infer the number of page references. This solution is probably less efficient that the first alternative, but, it does not require creation of special purpose histograms.

In our experiments, we used the second alternative, as outlined next. Suppose that a filter can have M pages and that the index used is clustered on the join column. We use a histogram on the outer's join column. To create a join filter that includes M pages we proceed as follows:

1. Find the histogram bucket with the (next) largest height.

2. Include the bucket bounds into the filter, extended up to the nearest page boundaries.

We repeat these steps until the size of the join filter is M pages. The bucket extension in step 2 makes the filter more efficient by including the tuples that share the same page. We note that filter construction is fast since it based on histograms, which are small in size. An example of a histogram and a corresponding join filter is shown in Figure 4.

To create a filter for an unclustered relation, one would need to consult the histogram and an unclustered index to see which pages are frequently referenced. Alternatively, one could only consult the histogram and assume that each tuple is on a separate page.

In general, a join filter can be expressed as a disjunct of range selections. In practice, since histograms assume uniform frequency within a bucket, many entire buckets will be included in a filter. Therefore, the size of a filter will be about the same size as a histogram. Since histograms are typically small, testing for filter satisfiability in step 2 of a Filtered INL join phase is quick. Our techniques can be easily adapted to any other statistical summary.

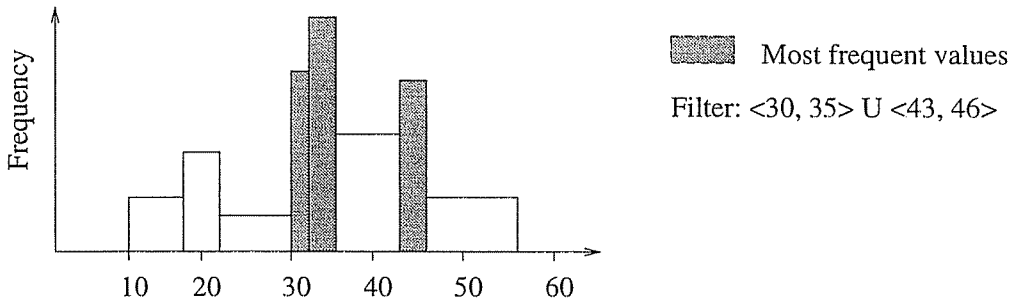

Figure 4: A histogram and the corresponding filter.

2.1.2 I/O Cost For Filtered Index Nested Loops Join

In this subsection we derive an analytical expression for the I/O cost for the first phase of a Filtered INL join, and then generalize it to include all phases. Since filtered joins will be used when optimizing for N answers, the cost formula must be a function of N, the target number of answers. The input parameters that determine the cost are:

M, the number of memory pages allocated to the inner relation, which is equal to the size of the filter.

h(M), the selectivity of the current join filter on the outer relation, it depends on the size of the filter and on the join column distribution.

|S|, the number of pages in the outer relation.

{S}, the number of tuples in the outer relation.

We first estimate the number of I/Os on the inner relation in the first phase of a join. This number is at most M, however it depends on N. Initially, almost all index lookups will result in I/O operations. Later on, there will be almost no I/O, as all the pages satisfying the join filter will be in memory. Denote the number of I/Os on the inner relation by P_R. For each output tuple produced, the probability of an I/O on the inner relation is join value dependent. Pages containing more frequent join values are more likely to be in memory than other pages, assuming the LRU replacement policy. At this point we use the simplifying assumption that any tuple within the filter has the same frequency of occurrence in the outer relation. This assumption is a reasonable approximation, as experimentally verified in Section 3.6 , and it makes derivation of an analytical expression possible. Under this assumption, the likelihood of a page fault on the inner relation when an output tuple is produced is: $\frac{M-P_R}{M}$. That is, there are $M - P_R$ unoccupied slots in memory of size M, and referring to any of them will result in a page fault. This can be expressed as a differential equation:

$$\frac{dP_R}{dN} = \frac{M - P_R}{M}$$

It can be easily checked that the solution for this equation is:

$$P_R = M(1 - e^{-\frac{N}{M}})$$

As discussed before, for a small N, $P_R \approx N$, while for large N, $P_R \approx M$.

Next, we estimate the number of I/Os for the outer relation. To produce N output tuples we need to read $N/h(M)$ outer tuples (as only the ones satisfying the filter are passed on). The number of I/Os per each outer tuple is $|S|/\{S\}$. It follows that the number of I/O operations on the outer relation, P_S, is:

$$P_S = \frac{N}{h(M)} \frac{|S|}{\{S\}} \tag{1}$$

This brings the total number of I/Os in the first phase ($P = P_R + P_S$) to:

$$P = M(1 - e^{-\frac{N}{M}}) + \frac{N}{h(M)} \frac{|S|}{\{S\}} \tag{2}$$

The maximum N produced in the first phase of a filtered join is $\{S\}h(M)$. If a larger N is needed the second join phase must be performed, which is identical to the first phase, only with a different join filter. The total cost is the sum of the costs of all the join phases. In practice, the number of phases will be small, because a blocking operator is likely to be more efficient than a multi phase Filtered INL join.

2.1.3 Optimal Join Filter

Our underlying objective is to minimize the cost to produce N answers by a join operator. We approximate the cost measure by the number of I/O operations; the inclusion of CPU cost would be straightforward. The question we consider is this: What join filter should be used to minimize the number of I/O operations, given that all other parameters are fixed? It is easy to see that Eq. (2) is minimized when $h(M)$ is maximized. Finally, we clearly see the reason for selecting the most frequent join values in the outer relation—such a filter minimizes the cost of a filtered join, under the given memory constraints.

2.1.4 Optimal Memory for a Filtered Join

In a typical database system, memory is a precious resource and it is therefore important to know what the maximum memory requirement of a join operation is. To find the optimal memory needed by a filtered join, we need to minimize Eq. (2) with respect to M. To find the minimum of the cost function (Eq. (2)) we can use a standard function minimization algorithm such as Golden Section Search [20]. Of course, the minimum memory required by a Filtered INL join is just one page ($M = 1$).

2.2 Multiple Foreign Key Joins

If individual joins comprising the execution pipeline are independent from each other, then maximizing the tuple rate for each join will maximize the output rate of the query. However, if correlations exist between join columns in different joins, then creating filters to maximize each individual join rate may *not* maximize the overall query rate , as illustrated by the following example.

Example 2.1 Consider a three way foreign key join of tables: **A** with two columns A_1 and A_2, shown in Figure 5, **B** with one column, shown in Figure 6, and **C** with one column, shown in Figure 7. Join conditions are $A_1 = B_1$ and $A_2 = C_1$. Notice that there is a large

A_1	A_2
1	1
1	2
1	3
2	4
2	4
3	5
4	5
5	5

Figure 5: Table A.

B_1
1
2
3
4
5

Figure 6: Table B.

C_1
1
2
3
4
5

Figure 7: Table C.

correlation between columns A_1 and A_2; high values of A_1 are associated with high values of A_2. Suppose that a join filter may have only one tuple. Considered independently, the join filter on table B would consist of value 1 (the most frequent A_1 value) and the join filter on table C would consist of value 5 (the most frequent A_2 value). However, the pair (1, 5) does not even exist in relation A, and therefore these two filters combined would produce no results! To create a good filter, one should filter pairs of B and C values that are the most frequent in relation A. In this case it is the pair (2, 4). □

If multiple join attributes are correlated, a multidimensional join filter must be produced for such a group of attributes. A multidimensional join filter is comprised of tuples that have the most frequent *combinations* of correlated attributes. To capture attribute dependences, we need multidimensional histograms [15]. The technique for constructing and using multidimensional filters is the same as the one for single attribute filters. Hence we do not discuss the correlated join case further in this paper.

3 Experimental Evaluation of Partial Join Execution

3.1 Testing Environment

We ran our experiments on a NT 4.0 workstation with 128 MB of memory and a 400 MHz Intel Pentium II processor. To measure the impact of filtered joins in an unbiased environment we decided to use a commercial database system rather than a simulation or an experimental system. We conducted our experiments on an IBM's DB2 Universal Database version 6.1, configured with 4KB pages. To eliminate buffering by the NT file system, we set the DB2 registry variable NTNOCACHE = yes. This way we could control the amount of memory used by the system by altering the size of the DB2 buffer pool. The amount of memory available to all algorithms was the same.

3.2 Join Methods Compared

1: Index Nested Loops (INL) join. A clustered index was built on the foreign key table. To make the DB2 select an INL join we used the OPTIMIZE FOR N ROWS clause. We used N = 10, but any small number would have the desired effect.
2: Filtered Index Nested Loops join. Since we could not modify the join processing of the DB2, we achieved the effect of filtering by rewriting the SQL query with additional

filter predicates. Actual values of the filter predicates were obtained from an externally maintained histogram. For example, to simulate the effect of a filter with the most frequent values in the interval from 10 to 20, we can rewrite the join query from Example 1.1 as follows:

```
select price, manufacturer from sales s, products p
where s.productId = p.productId and s.productId between 10 and 20
```

Before loading the data into the database, we built a compressed histogram [14], which is an enhanced version of the widely-used equi-depth histogram. The size of a histogram was fixed to a typical value of 500 Bytes. (Using a larger histogram, or higher quality algorithms, such as V-optimal [14] or Wavelets [16], would produce even better filters.)
3: Sort merge join. This represents the best DB2 timing when the query is optimized for all answers.

3.3 Data Sets

3.3.1 Real Data Set

We used point of sale data of a Japanese drugstore chain Pharma [7], consisting of two tables: Sales and Products (introduced in Example 1.1). To execute the query from Example 1.1, the system needs to perform an equi-join on *productId* between Sales and Products. The cardinality of Products and Sales are 27,215 and 893,402, while the tuple sizes are 96 and 110 bytes respectively. Since it was not possible to determine the number of inner tuples per page (the DB2 manual provides only an approximate formula), our simple policy is that if a tuple is selected to be in a filter, at least 10 neighboring tuples are selected as well. This reflects an assumption that 10 neighbors of a particular tuple are likely to be on the same page, which is reasonable since the number of inner tuples per page is 40. Performance results are not very sensitive to this number, as long as it is much smaller than the number of tuples per page, and the results are good enough to illustrate the gains of filtering.

3.3.2 Synthetic Data Set

Conceptually, in terms of tables and schemas, the synthetic data set is the same as the real data set. However, it differs in the distribution of the join values, which were varied throughout the experiment.

Join attribute values form clusters, which are positioned according to Zipf distribution [22]. Zipf distribution is known to accurately model skew in real-life data. Each cluster is bell shaped, with the same standard deviation (width) but with a variable number of points (size). The number of points within a cluster was also distributed according to a Zipf distribution. The summary of parameters, along with the default values used in experiments, are shown in Table 1.

The cardinality of the outer and inner relations are 1,000,000 and 100,000 respectively, while the tuple sizes are 100 bytes in both relations.

Parameter	Values
Number of clusters	500
Skew in the cluster sizes	1
Skew in the cluster positions	0
Standard deviation within a cluster	100
Total distribution width	100,000

Table 1: Synthetic data parameters with values.

3.4 Experimental Results with Real Data Set

3.4.1 Time Dependency

We measured times of answer tuples arrivals for the query from Example 1.1. Results, when the buffer size is 250 pages, are shown in Figure 8. The horizontal line represents the Sort Merge join, characterized by a pause of about 80 seconds before the first answer tuple is delivered. This is unacceptable for an interactive user who may prefer to obtain some answers immediately. As shown by the dashed line, the INL join achieves this. However, in case users are interested in seeing any significant fraction of the complete result set (more than 4% in this case) they will have to wait for a much longer time than that required for the Sort Merge join. The Filtered INL join has desirable characteristics of both previous methods: quick response time and efficient delivery of a larger number of answers. We note that some items in the sales data are sold much more frequently than others. Filtered join takes advantage of this fact by retaining such items in memory, which results in a large performance advantage.

3.4.2 Memory Dependency

We vary the number of memory pages allocated to the join and show the time needed to produce 6% of the answer set in Figure 9. This figure demonstrates that histogram filtering is effective over a large range of memory sizes (up to the full size of inner relation). However, the advantage over traditional INL join is less pronounced for higher memory sizes. This is expected since the INL join causes much less I/Os when given enough memory.

3.5 Experimental Results with Synthetic Data Sets

3.5.1 Time Dependency

Early tuple delivery rate is shown in Figure 10 and it looks much like the corresponding figure for the drugstore data (Figure 8). Filtered INL join runs significantly faster than the traditional version, since the most frequently referenced tuples are found in the buffer pool. The Sort Merge join blocks for about 120 seconds but delivers answers rapidly from there on. Due to excessive thrashing, the execution time for an INL join exceeds that of the Sort Merge join after 1% of the answer tuples are returned.

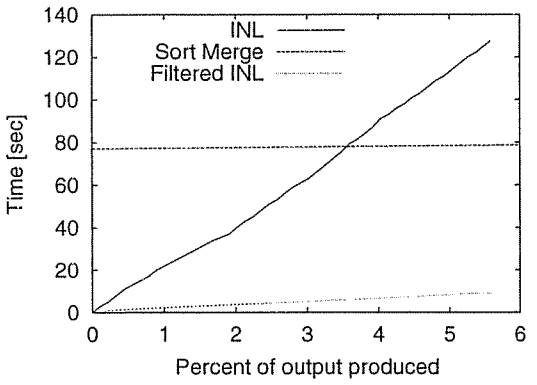

Figure 8: Elapsed time vs. percent of output produced (real data set).

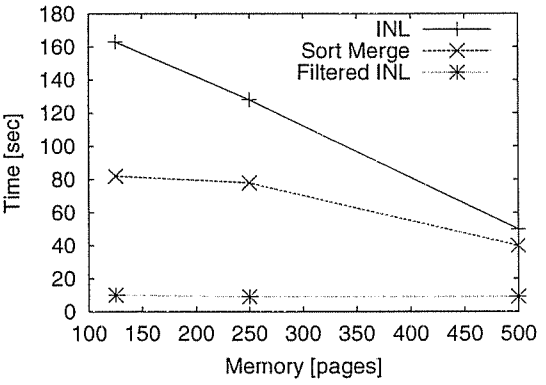

Figure 9: Time to produce 6% of output vs. buffer pages (real data set).

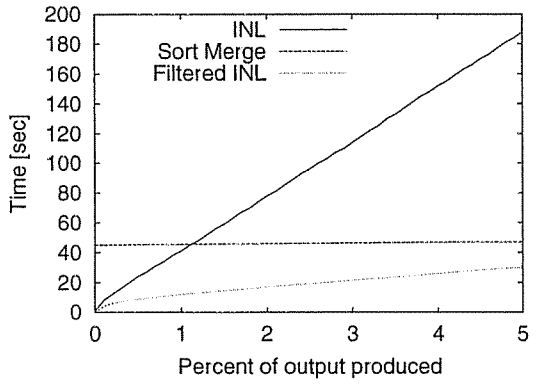

Figure 10: Elapsed time vs. percent of output produced (synthetic data set).

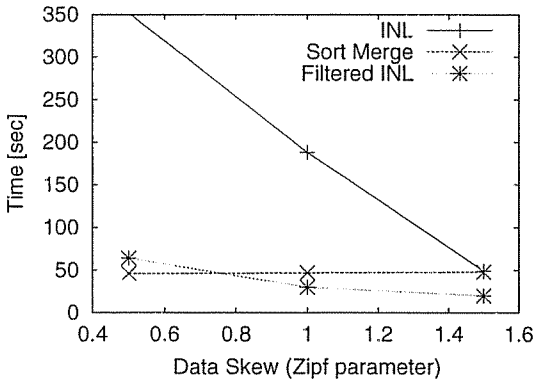

Figure 11: Time to produce 5% of data vs. data skew (synthetic data set).

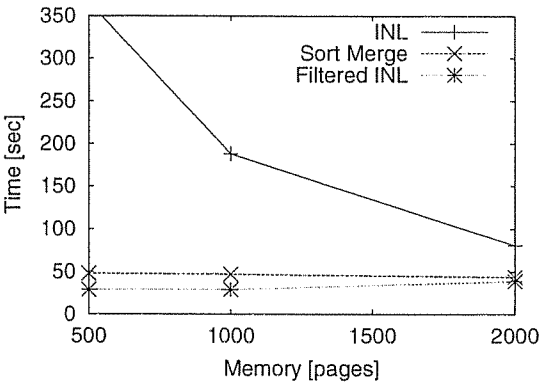

Figure 12: Time to produce 5% of data vs. buffer size (synthetic data set).

3.5.2 Skew Dependency

In Figure 11 we show the time needed to produce 5% of the answers for different skews of the cluster sizes in the outer relation. The differences between algorithms are less pronounced for the higher skew because the LRU replacement policy of the buffer manager effectively keeps the most frequent values in the cache, the same task that is achieved by the join filter.

3.5.3 Memory Dependency

In Figure 12 we show the time needed to produce 5% of the answers for different buffer sizes. This figure demonstrates that differences between filtered and traditional joins are more pronounced when less memory is available. The same trend was observed and explained in the real data set experiment (Section 3.4). It is interesting to notice that the Filtered INL join may not always be faster when given more memory. This emphasizes the importance of finding the optimal memory by minimizing Eq. 2 as described in Section 2.1.4.

3.6 Simulation of Filtered Join I/O Operations

In this section we verify the plausibility of an approximation used in Section 2.1.2. We compare the total number of I/Os predicted by formula (2) versus the simulation results of the synthetic data set described in Section 3.3.2. The results, presented in Figure 13, show that the two curves are close enough; the discrepancy is mostly confined to the early stages of the join while the memory is being loaded with the inner tuples that satisfy the join filter. The total number of I/Os is dominated by the outer relation, whose contribution is linear in the number of answers produced. (See Eq. 2). Notice that the number of I/Os on the inner relation is bounded by M, the memory size, so once all the inner tuples are brought in memory (1,000 pages in this case), the cost curve becomes a straight line, as evident from Figure 13.

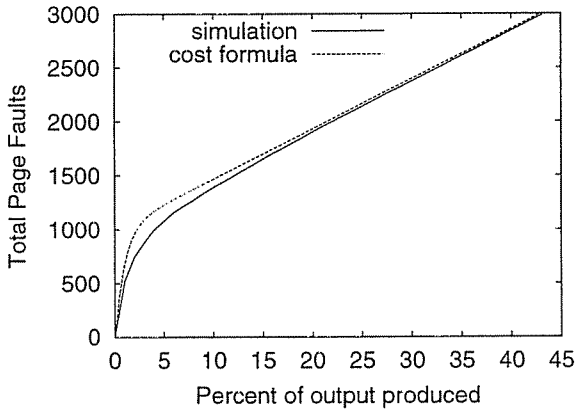

Figure 13: Percent of output produced vs. total I/O.

4 Optimizing Queries For N Answers

As we have seen from the experimental section (Section 3), filtered joins are faster than standard join methods when a small fraction of result tuples are required. Consequently, an optimizer should create plans involving Filtered Index Nested Loops joins when optimizing for N answers. Current query optimizers, however, optimize for all answers, so we describe how a standard, dynamic programming based, optimizer [17] can be modified to optimize queries for N answers. We note that the best plan for N answers must be identical to the best plan for all answers when N is very large. We review some basic concepts in query optimization next.

A query can be represented as a tree of relational operators such as select, project, and join. Such relational operators are called *logical operators*. *Physical operators* are specific algorithms to evaluate logical operators, such as Hash join, Index scan, etc. A *plan* is a tree of physical operators. There are many plans that are logically equivalent but differ in cost. A traditional optimizer takes a tree of logical operators and generates a logically equivalent plan with the minimum cost to produce *all* answers.

In this section we focus on join optimization, since it is the most difficult aspect of query optimization. A standard optimizer builds join plans bottom up, starting from one relation subplans. It then uses already generated subplans as building blocks for larger plans. If two logically equivalent plans are generated, the more expensive one is eliminated,i.e., it is not considered as a part of any larger plan.

As indicated before, our task is to modify an optimizer to generate plans with minimum cost for obtaining N answers. We propose to maintain plans optimized for partial answers, in addition to traditional plans optimized for all answers. To distinguish between two types of plans, we mark each plan as either (1) optimized for all answers or (2) optimized for partial answers. For plans optimized for partial answers, we introduce a new physical property called *optimization cardinality*; it is the number of answers that a plan is optimized for. The optimization cardinality N_X for a plan X must be such that, on average, N query answers would be produced if the cardinality of X was N_X. Let T be the cardinality of the query result, and T_X be the cardinality of the plan X. Assuming that the tuples contributing to the final answers are uniformly spread throughout table X, we have $\frac{N_X}{N} = \frac{T_X}{T}$, from which

it follows that:

$$N_X = N\frac{T_X}{T}. \tag{3}$$

Note that the optimization cardinality may be any number between 1 and the total plan cardinality.

In addition to traditional optimization rules, the following rules are needed for partial query optimization:

1. We need to create all possible single relation plans and mark them as optimized for partial answers. The optimization cardinality of these plans is given by (3). To evaluate the cost of a single relation plan for partial answers, we count only the work needed to produce the number of answers given by (3). For example, the cost of a File scan plan with optimization cardinality 1 is just 1 I/O.

2. To build a larger plan using a blocking join operator, both input plans must be optimized for all answers. This is because a blocking operator, by definition, requires both of its *entire* inputs before any answer tuple can be produced. We mark the resulting plan as optimized for all answers. [1]

3. When building larger plans using Nested Loops joins, the inner plan must be optimized for all answers, since the entire inner relation is required before any answer tuple may be produced. If the outer input is marked as optimized for all answers, we mark the resulting plan the same. If the outer input is marked as optimized for partial answers, we proceed as follows: (1) Evaluate the cost of the join as if the cardinality of the outer subplan is equal to its optimization cardinality, (2) determine the resulting optimization cardinality using Eq. (3), and (3) mark the resulting plan as optimized for partial answers.

4. If a plan optimized for a partial number of answers is more expensive than a logically equivalent plan optimized for all answers, it may be eliminated. This is due to the fact that the cost of a plan may only decrease with the smaller optimization cardinality. Note that a plan optimized for all answers may not be eliminated, even if it is more expensive, since more efficient blocking algorithms are applicable to such plans.

To illustrate the search process when optimizing for N answers we present the following example.

Example 4.1 Consider a three way join between tables A, B, and C, as described in Example 2.1. Suppose that a plan optimized for 10 answers is needed. Single table subplans considered by an optimizer are A, B, and C. (We use table names to denote subplans, at the risk of confusing tables and plans, however the meaning is always clear from the context.) Because we need to optimize for 10 final answers, an optimizer will also consider plans A', B', and C' where the prime (') stands for a subplan optimized for partial answers. Since both of our joins are foreign key joins, the optimization cardinality for all single relation plans is 10, the final optimization cardinality (according to Eq. (3)). Two table subplans considered by an optimizer are AB and AC (since we don't consider cross products) and

[1] We could also cost a blocking join for a partial number of answers but the savings will be small since most of the execution time is the "think" time.

the corresponding (AB)' and (AC)' plans. The best plan for group AB is the cheaper of AnyJoin(A, B) and AnyJoin(B, A). Possible solutions for (AB)' is NestedLoops(A', B) and NestedLoops(B', A), according to Rule 3. If (AB)' is more expensive than AB, it is pruned (not considered as a part of any larger plan), according to Rule 4. Analogous statements are true for plans AC and (AC)'. The final plans are ABC and (ABC)'. The best plan for the group ABC is the cheaper of AnyJoin(AB, C) and AnyJoin(AC, B), since we consider only the left-deep trees. The best plan for group (ABC)' is the cheaper of NestedLoops((AB)', C) and NestedLoops((AC)', B). The final plan is the cheaper of ABC and (ABC)'. □

5 Using Filters to Reduce Total Execution Time

In this section, we demonstrate that using statistics may also reduce the total execution time of blocking operators. We achieve significant cost reduction in Hybrid Hash join and group-by operators when the distribution of the join column is skewed.

5.1 Hybrid Hash Join

Hybrid Hash join algorithm (see [18] for an overview) works by partitioning inner and outer relations. In the first phase of the algorithm, the inner (smaller) table is partitioned into a number of memory sized partitions, which are written to temporary files on disk, and one memory resident partition. During the second phase, the outer table is hashed and tuples that hash into the memory resident partition are immediately probed and joined, while the other tuples are written to their partitions on disk. In the third phase, the matching disk resident partitions are joined. Since the outer tuples that hash into disk resident partitions have to be written to disk and read back in, the more tuples that hash into the memory resident partition, the smaller the number of I/O operations. To reduce the I/O, the algorithm uses all available memory in the first phase to make the memory resident partition as large as possible.

Consider a foreign key join where the foreign key table is chosen as inner. To save on I/O in the outer table, the number of outer table tuples that hash into the memory resident partition should be maximized. This means that the candidate tuples for the memory resident partition should have the most matches in the outer table. Therefore, we suggest using database statistics to find the most frequent values of the join column in the outer table and storing these values in the memory resident partition. We illustrate our strategy in Figure 14. The figure shows traditional Hybrid Hash join partitioning, which is based on hash values and therefore uniform, and Filtered Hybrid Hash join partitioning, which is characterized by a large first partition of the outer table, since it contains the most frequent join values. Because the first partition of the outer table is not written to disk, the larger it is, the smaller the number of I/O operations. We demonstrate gains of this strategy by analyzing the cost formula for Hybrid Hash join.

Let R denote the build relation and S the probe relation in a Hybrid Hash join algorithm and let $|R|$ and $|S|$ denote the respective number of pages while $\{R\}$ and $\{S\}$ denote the respective number of tuples. Furthermore, let $MIPS$ denote the CPU speed of the machine, $T_{I/O}$ denote the average I/O service time, ψ denote the selectivity of the join, while f_R and f_S denote fractions of the respective relations that belong to the memory resident partition.

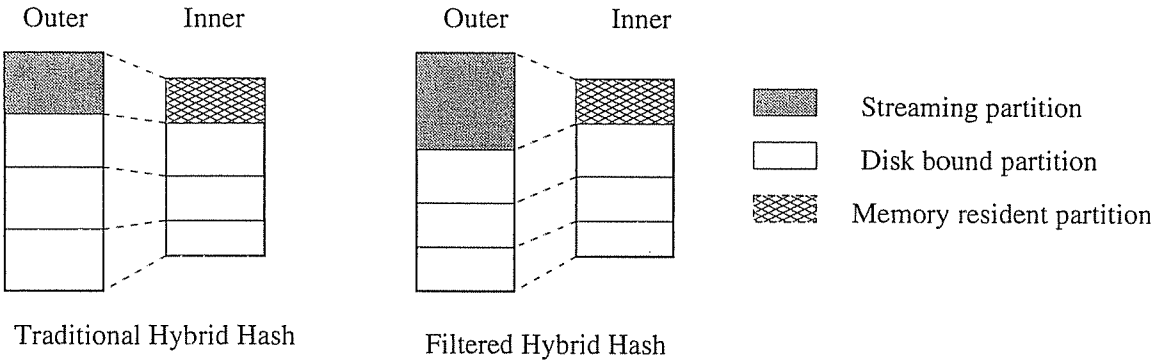

Figure 14: Comparison of traditional vs. Filtered Hybrid Hash join.

Parameter	Value
$\|R\|$	25,000
$\{R\}$	100,000
$\|S\|$	25,000
$\{S\}$	10^6

Table 2: Data parameters.

Parameter	Value
T_{IO}	10 ms
MIPS	500 MHz
I_{move}	500
I_{join}	100
I_{hash}	100
I_{search}	1000
$I_{I/O}$	3000

Table 3: Cost model parameters.

Then the total cost of the Hybrid Hash is [21]:

$$\text{I/O Cost} = T_{I/O}[(3-2f_R)|R| + (3-2f_S)|S|], \tag{4}$$

$$\begin{aligned}\text{CPU Cost} = \frac{1}{MIPS} * \{&I_{I/O}[(3-2f_R)|R| + (3-2f_S)|S|] + \\ &I_{move}[(3-2f_R)\{R\} + (3-2f_S)\{S\}] + \\ &I_{join}\psi\{R\}\{S\} + I_{hash}[(2-f_R)\{R\} + (2-f_S)\{S\}] + \\ &I_{search}(\{R\} + \{S\})\},\end{aligned} \tag{5}$$

where various I's stand for the instruction path lengths for corresponding operations. The value of f_R is $M/|R|$, where M is memory allocated to the join. Obviously, this formula is a decreasing function of f_S, which is maximized when the memory resident partition of R contains join attribute values that are the most frequent in S.

For an experimental evaluation of these ideas we used synthetic data generated as described in experimental section (Section 3) with data parameters shown in Table 2 and cost parameters shown in Table 3. The memory given to the join operator is 40 MB. In general, the smaller the memory the larger the impact of filtering. Results of experiments are shown in Figure 15. As expected, using a filter to define the content of the memory resident partition is especially beneficial for high skew data, where the time saving is about 25% for this typical data set. This is because a large percentage of the outer relation is matched to the memory resident partition.

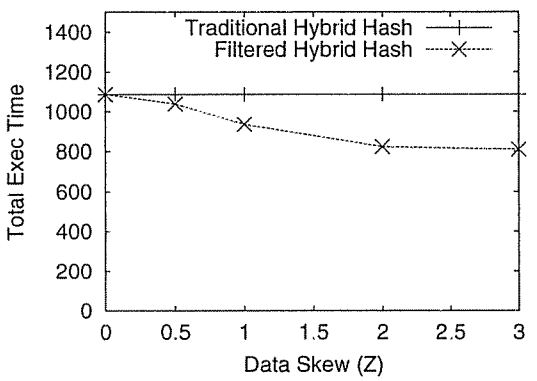

Figure 15: Total execution time for Hybrid Hash join vs. data skew.

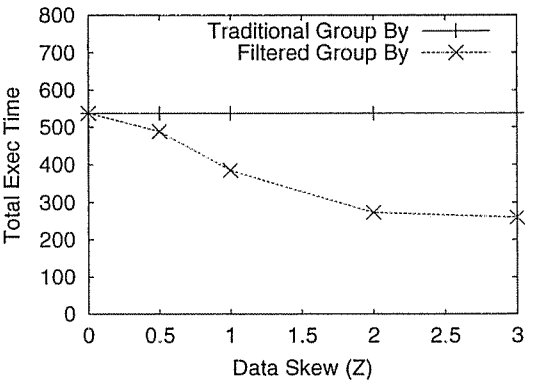

Figure 16: Total execution time for group-by operator vs. data skew.

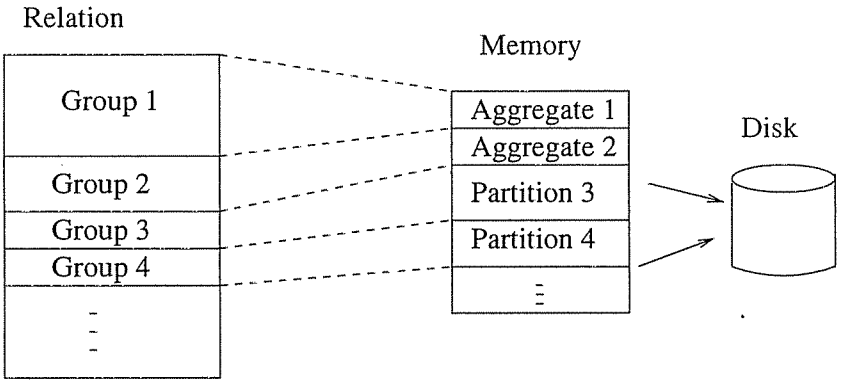

Figure 17: Filtered hash-based group-by operator aggregates the largest groups in memory.

5.2 Hash-Based Group-By Operator

A hash-based implementation of a group-by operator is analogous to the Hybrid Hash algorithm. Results for a certain number of groups are aggregated in memory while the tuples from other groups are hashed to appropriate partitions that are then grouped in memory during the second pass. We propose to determine the largest groups from database statistics and make sure that such groups are aggregated in memory. This strategy, which is illustrated in Figure 17, minimizes the number of tuples passed to disk. The exact cost formula for group-by operator is a simplified version of Equations (4), and (5) with no R and I_{join} terms. The experimental results are shown in Figure 16. It can be seen that relative benefits of filtering are even larger here than in a Hybrid Hash join since there are no filter independent R terms. We note that our technique can reduce the execution time in half for this typical data set. Memory allocated to the group-by operator is 10 MB.

6 Related Work

Optimizing queries for interactive response time is a topic of considerable interest for the database community. To achieve interactive response time for select-project-join queries,

some commercial systems have extended SQL with a clause that instructs an optimizer to minimize the time to produce the first N answers. Examples are IBM's DB2 Universal Database OPTIMIZE FOR N ROWS clause [2], and Microsoft SQL Server 7.0 [3] OPTION FAST N clause. When this clause is present, the optimizer typically chooses a pipelined plan, provided that the value of N is small enough. There is no publicly available documentation on how the feature was implemented, however, we obtained some high level information from [11] and [5] that indicates certain similarities with the approach outlined in Section 4.

Optimization for the first answers is considered in [9], however, the main contribution was to reduce wasted effort in join pipelining. Authors observed that a pipeline of reductive join operators could be more efficient if the next tuple was always drawn from the first join operator whose tuple failed to match. We, on the other hand, consider foreign key (non-reductive) joins for which this work does not apply. We are also primarily concerned with a single join not a pipeline of joins, even though our work applies to a series of joins as well.

Ideas about getting certain number of query answers quickly were also presented in [19]. The emphasis of this paper is on heuristic query rewriting and decomposition into subqueries that can be evaluated on demand. The decomposition strategy is based on system configuration and past user behavior.

Online aggregation framework [8, 6] is closely related to our work since the objective is to achieve interactive response times. However, the focus of the work on online aggregation (and in particular Ripple joins) is to produce output with desired (random) statistical properties. In contrast, our work is focused on producing answers in the shortest possible time.

Previous work on partial query evaluation ([13, 4, 1]) is largely orthogonal to ours. The TOP N operator, which is the main focus of these papers, is conceptually a sequence of operators starting with SORT and followed by OPTIMIZE FOR N and STOP AFTER N. In fact, our work is not concerned with SORT operator because the problem would reduce to the TOP N query.

A distinguishing feature of our work is the use of system statistics to guide query execution. This idea was not proposed nor evaluated in any of the previous work, to the best of our knowledge.

7 Future Work

One of the directions for future work is to explore potential gains that filtering may offer in a wide-area network, where the bandwidth is scarce. For example, when transferring a relation over a slow wide area network in distributed join processing, one might first ship data satisfying the join filter. This way, we use the scarce bandwidth efficiently, as we first ship the data that will produce the maximal number of answers. This strategy is similar in its nature to bloom filters [12].

More research needs to be done to address the issue of answer correlation when using filters in answering partial queries. Users see the results that satisfy the filter but are not aware of filter existence and its influence on the early results. In other words, first tuples produced by a filter are not a random sample of the total answer set and such an assumption may mislead the user. A solution to this problem is to verify that the initial answers are random by maintaining the information about attribute correlations. For example, if a

query asks for attribute A and the filter is on attribute B, the filtering technique will not impose any specific ordering of answers only if there is no correlation between A and B.

8 Conclusion

We explored the idea of using system statistics to improve the execution speed of major relational operators such as join and group-by. Our experimental results indicate that significant performance gains are possible in both partial and full execution of those operators. Filtered join and group-by algorithms can be easily incorporated into a traditional cost based database optimizer, and can naturally be included into an online aggregation system. We believe that histogram filtering is an important step towards achieving interactive response times for complex, ad hoc, decision support queries.

References

[1] Surajit Chaudhuri and Luis Gravano. Evaluating top-k selection queries. In *Proceedings of VLDB Conference*, pages 397–410. Morgan Kaufmann, 1999.

[2] IBM Corporation. *IBM DB2 Universal Database Administration Guide (version 6)*. 1999.

[3] Microsoft Corporation. *Administering SQL Server 7.0*. 1999.

[4] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In *Proceedings of VLDB Conference*, pages 411–422, 1999.

[5] Goetz Graefe. *Microsoft Corporation*. Private Communication.

[6] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 287–298. ACM Press, 1999.

[7] Yukinobu Hamuro, Naoki Katoh, Yasuyuki Matsuda, and Katsutoshi Yada. Mining pharmacy data helps to make profits. In *Data Mining and Knowledge Discovery 2(4)*, pages 391–398, 1998.

[8] Helen J. Wang Joseph M. Hellerstein, Peter J. Haas. Online aggregation. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, Tucson, Arizona, 1997.

[9] Roberto J. Bayardo Jr. and Daniel P. Miranker. Processing queries for first few answers. In *Proceedings of International Conference on Information and Knowledge Management*, pages 45–52, 1996.

[10] Miron Livny, Raghu Ramakrishnan, Kevin S. Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and R. Kent Wenger. Devise: Integrated querying and visualization of large datasets. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 301–312, 1997.

[11] Guy M. Lohman. *IBM Corporation*. Private Communication.

[12] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proceedings of VLDB Conference*, pages 149–159, 1986.

[13] Donald Kossmann Michael J. Carey. On Saying "Enough Already!" in SQL. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, Tucson, Arizona, 1997.

[14] Viswanath Poosala, Yannis Ioannidis, Peter Haas, and Eugene Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 294–305, June 1996.

[15] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of VLDB Conference*, pages 486–495. Morgan Kaufmann, 1997.

[16] Jeffrey ScottVitter, Min Wang, and Bala Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of International Conference on Information and Knowledge Management*, 1998.

[17] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, pages 23–34, 1979.

[18] Leonard D. Shapiro. Join processing in database systems with large main memories. In *ACM Transactions on Database Systems*, volume 11, pages 239–264, 1986.

[19] Kian-Lee Tan, Cheng Hian Goh, and Beng Chin Ooi. On getting some answers quickly, and perhaps more later. In *Proceedings of IEEE Conference on Data Engineering*, 1999.

[20] Saul A. William H. Press, Brian P. Flannery and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1993.

[21] Philip S. Yu and Douglas W. Cornell. Buffer management based on return on consumption in a multi-query environment. *VLDB Journal*, 2(1):1–37, 1993.

[22] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley, Reading, MA, 1949.