Buffer Pool Aware Query Optimization

Ravishankar Ramamurthy David J. DeWitt

Department of Computer Sciences,
University of Wisconsin-Madison
Madison,
USA
ravi@cs.wisc.edu, dewitt@cs.wisc.edu

Abstract

With the advent of 64-bit processors, large main memories are set to become very common. This in turn translates to larger buffer pool configurations in database servers. Query optimizers however, currently assume all data is disk resident while optimizing queries. This assumption will no longer be valid when buffer pools become 100's of gigabytes in size. In this paper we examine how data presence in the buffer pool can affect the choice of query plans in an optimizer. We examine the possible benefits of buffer-pool aware query optimization and propose a generic architecture for implementing such an optimizer.

1. Introduction

While the basic approach to query optimization has not changed since 1979 [21], the rest of the environment in which database systems operate has changed dramatically. Processors are 1000 times faster. Memories and disks are 1000 times bigger. Query execution techniques have also improved dramatically with more efficient algorithms, techniques such as bit-mapped and covering indices, materialized views, and parallel execution. Improvements have certainly occurred in optimizers including the use of histograms to estimate selection cardinalities [20] and rule-based techniques for rewriting complex queries [12]. However, the basic paradigm of query optimization has gone unchanged: the optimizer explores a number of plans, estimating the cost of each, and picks what it thinks is the best plan, which is then executed. While this strategy worked very well when the query engines were not capable of executing queries with more than a couple of trivial join operators in a reasonable amount of time, it no longer works very well today with queries involving dozens of very large tables. There are several fundamental problems. Foremost is the problem of error estimation in join cardinalities [13]. Basically, after 1-2 join operators, it is impossible for an optimizer to estimate join cardinalities accurately. This makes picking the best join order or join method for subsequent joins in a complex query essentially impossible. The bottom line is that our ability to execute increasingly complex queries over very large data sets has increased at a much faster rate due to improvements in hardware and software than our ability to optimize such queries correctly. Over the last couple of years, several attempts have been made to improve the state of query optimization. One approach is a technique known as dynamic query optimization in which optimization and execution are interleaved [2, 3, 14, 15]. The idea actually dates back to the INGRES [24] project. Leo [22] represents another approach for improving query optimization. Here the idea is to use statistics gathered at run-time to improve the optimizer's statistics.

This paper considers another technique for improving the effectiveness of query optimization. Currently no optimizer that we are aware of considers the contents of the buffer pool when optimizing a query. Optimizers always assume that all tables at the leaves of the query plan are disk resident, and ignore the contents of the buffer pool when evaluating alternative execution plans. Even though it is generally safe to assume there is not enough main memory to hold the entire database, at any given point a significant fraction of the active tables may be memory resident.

Ignoring the contents of the buffer pool while optimizing queries can cause the optimizer to pick sub-optimal plans. Consider the following example. Assume a query includes a selection predicate on a table and assume that there is an

Proceedings of the 2005 CIDR Conference

unclustered index available on the attribute on which the predicate is defined. Hence, the optimizer can choose to either use an unclustered index scan or sequentially scan the entire table in order to execute the query. Beyond some threshold value in predicate selectivity, the optimizer will almost always pick the sequential scan. However, if the data pages that are actually accessed when evaluating the predicate are already resident in the buffer pool, using the index scan will be faster. Data cached in the buffer pool can also affect other choices made during query optimization including join ordering and the selection of a join algorithm.

The cost per MB of main memory has dropped by a factor of 10,000 over the last twenty years [11]. A rule of thumb suggested in [10] is that data that is stored on disk today will be stored in main memory in ten years. Techniques like vertical partitioning [7][4] may further enhance the use of large amounts of main memory as a cache for frequently referenced columns. In this paper we examine how this trend is likely to affect the way query optimizers and query engines are architected. The situation we are considering is not a main memory database system, which typically assumes that the entire database fits into main memory. There are several commercial main memory database products like TimesTen [26] available today. With storage costs' decreasing rapidly, one possibility is that main memory database servers may replace traditional database servers. We do not think this scenario is likely. Main memory databases are currently used for specialized applications where the database size limits can be guaranteed or as a high-speed cache for a traditional relational DBMS in web applications. As more non-traditional data like images and historical data are incorporated into a database; it will probably not be cost effective to store the entire database in main memory. Thus, even though main memory databases are likely to remain important in certain niche environments, it is unlikely that they will be used for more traditional database applications like transaction processing and decision support.

At the other end of the spectrum, we have traditional databases that store data using a disk sub-system. Data pages are cached in a buffer pool as they are read from the disk sub-system using a suitable replacement policy. Thus, decreasing storage costs would imply a larger buffer pool that could cache more pages in memory. In fact, the "five-minute" rule [11] suggests that data pages that are accessed every five minutes should be memory resident. Given storage economics, this interval is likely to increase. However, just caching more pages in the buffer pool does not automatically guarantee improved performance. The goal of this paper is to demonstrate that adopting a query optimizer that is "buffer-pool aware" can significantly improve the overall performance of decision support queries. We are particularly concerned with exploratory environments where users issue a set of related queries and interactive response times are critical.

The remainder of this paper is organized as follows. Section 2 examines how the classical trade-off between using an unclustered index and a table scan varies as a function of the contents of the buffer pool. Section 3 extends this analysis to join queries. In Section 4, we introduce the concept of index pre-execution and outline a generic architecture for query processing that is "buffer-pool aware". The paper then presents its conclusions and suggests some interesting avenues for future work.

2. Single Table Queries

This section examines how the contents of the buffer pool can affect access path selection for a single table. In particular, we examine the choice between using a table scan and an un-clustered index scan.

2.1 Introduction

Query optimizers use cost functions to evaluate alternate evaluation plans. In choosing between a sequential scan of a table and an unclustered index scan this translates to some cut-off in predicate selectivity (say s_0). For predicates more selective than s_0, the optimizer should choose an unclustered index scan. Otherwise, a sequential table scan would be selected. An interesting point to note is that this threshold value is independent of the contents of the buffer pool. Over the course of executing queries, a significant number of pages that are required to answer a query might have become cached in the buffer pool. The relative performance of the alternative query plans can change based how many pages are resident in the buffer pool to the point that the optimizer can actually pick the wrong plan. In this section, we examine this problem in detail. Simple analytical formulae are used to analyze when a buffer pool aware optimizer is likely to be useful. Experimental results are then presented to show how buffer pool contents can affect the relative performance of a sequential scan vs. an unclustered index lookup.

2.2 Analytical Model

Consider a relation R and assume a hypothetical query workload that consists of a series of queries with selection predicates on R. Assume that an unclustered index exists on the attribute on which the selection predicate is defined. Thus, two evaluation plans for each query are possible; one that uses the index and the other that scans the table. A query optimizer would typically use a cost function to estimate the cost of each alternate plan and would pick the plan with the lowest cost. Assuming a simplistic cost model that considers only the I/O cost, the relevant parameters are:

- N, the number of pages in the relation R
- Nbuf, the size of the buffer pool in pages
- Tseq, the time to read a page using sequential I/O
- Trandom, the time to read a page using random I/O

Assume that Nbuf < N, i.e. the relation R will not fit completely in the buffer pool. Consider a query Q and assume that the predicate selects k records from the table. Then, the cost of the two alternate evaluation plans is:

Cost of Table Scan (cost1) = N*Tseq
Cost of Index Plan (cost2) = C(k)*Trandom

C(k) represents the Cardenas formula [25] for estimating the number of page accesses. If there are M records stored in N pages, the number of pages touched while accessing k records through an unclustered index is given by:

$$C(k) = N^* (1 - (1 - 1/N)^k)$$

Note that this expression is independent of M (the total number of records in the table). Yao mentions in [25] that the error involved in using this approximation is negligible for cases in which the number of tuples per page page (the blocking factor) is not a small number (say < 10). This assumption is typically true for pages sizes used in today's database systems (4 Kbytes–32 Kbytes). Moreover this formula is easier to manipulate mathematically than Yao's formula.

Traditional optimizers would choose the index plan as long as cost2 is less than cost1. The cut-off point (in terms of the number of records accessed) after which the optimizer would always pick the sequential scan occurs at the point at which the two cost functions intersect. This corresponds to the following equation.

$$N^*Tseq = Trandom^* C(k).$$

Denote Tseq/Trandom by d. (Note that d is always less than 1)

Substituting for C(k) we obtain:

$$N^*d = N^* (1 - (1 - 1/N)^k)$$

Simplifying, we obtain, $k = \log (1 - d)/ \log (1 - 1/N)$.
This represents the cut-off value in terms of number of records accessed after which the index scan should no longer be picked by the optimizer. Denote this value by k_0.

The cost functions used in the calculation above are independent of the contents of the buffer pool. Next we consider how the trade-off between the two plans can change if the optimizer takes into account the data pages cached in the buffer pool. Consider the following scenario. Assume that the database system has executed a number of queries, some of which access table R. Reconsider query Q. Assume that a fraction f of the pages holding tuples that satisfy the selection predicate on Q are already cached in the buffer pool as a result of executing other queries. The cost functions for the two alternate plans, given this knowledge would be:

Table Scan (cost1): $(N - f * C(k)) * Tseq$[1]
Index Plan (cost2): $C(k) * (1 - f) * Trandom$

Consider the cut-off point (in terms of number of records selected) under these revised cost estimates, after which the optimizer would pick the sequential scan. Consider the best scenario for the index scan; in this case all the pages in the buffer pool could be used to answer the current query (this can at most be equal to Nbuf). Thus, the cut-off point can be determined using the following equation:

$$(N - Nbuf)* d = (C(k) - Nbuf) \text{ or,}$$
$$N^*d + Nbuf (1-d) = N^* (1 - (1 - 1/N)^k)$$

Simplifying, we get
$k = \log ((1-d) * (1 - Nbuf/N)) / \log (1-1/N)$.

Let this value be denoted as k_1, which represents the selectivity value (in terms of the number of records selected) after which the optimizer would pick the scan given the knowledge of the contents of the buffer pool. Consider the expression (k1 – k0).

$$k1 - k0 = \log (1-Nbuf/N)/\log (1-1/N).$$

This is a positive number (recall that N > Nbuf). This represents the possible selectivity range (in terms of the number of records selected) in which a traditional optimizer would pick the scan and in which a "buffer-pool aware" optimizer would pick the index scan. This equation shows that there exist cases where the optimizer could choose the wrong plan unless it has knowledge of the contents of the buffer pool.

The previous analysis assumed the best case for the index scan (i.e. all the pages that are needed for the query are in the buffer pool). Consider the case in which the buffer pool just contains a random sample of the pages in relation R. In this case, the fraction of pages required to answer the current query that can be resident in the buffer pool can be at most (Nbuf/N) * C(k). The cut-off point in this case would be determined using the following equation.

$$(N - Nbuf) * d = C(k) (1 - Nbuf / N) \text{ or,}$$
$$N^*d = N (1 - (1 - 1/N)^k)$$

[1] For simplicity, we assume that caching does not affect the sequential bandwidth.

Simplifying, k = log (1-d)/ log (1 – 1/N) which is identical to k_0.

The results from the analytical models ascertain the following results that are fairly intuitive. Cost models that reflect the buffer contents are likely to make a difference when the workload has some definite locality. For workloads that just have a random footprint, a cost model that reflects the buffer contents does not provide any additional functionality. In this paper, we are looking at decision support applications where the workload is more likely to have some locality.

These formulae have demonstrated that there is a "grey" region in which an optimizer that does not exploit knowledge of the buffer pool contents can pick the wrong execution plan. However, it may not matter if the relative execution time between the plans in this region is negligible. In the following section the actual performance gains that can be achieved by exploiting buffer-aware optimization techniques are quantified experimentally.

2.3 Experimental Results

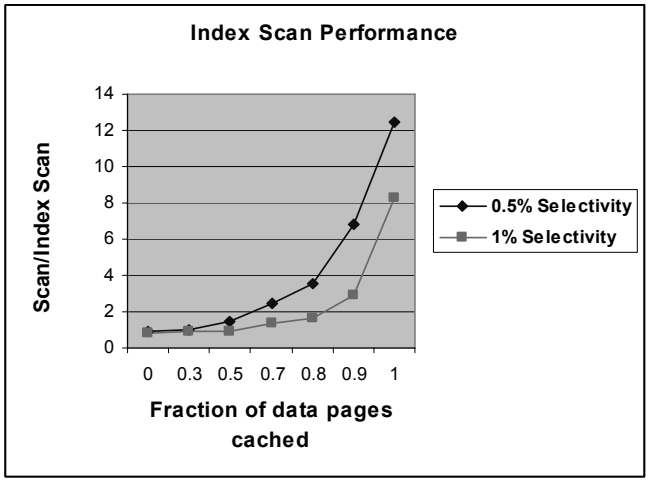
All the experiments in this paper were performed using a prototype relational query engine implemented on top of the SHORE storage manager [5]. The machine used is a Pentium 2 GHz processor with 1 GB of main memory running Red Hat Linux (9.0). The 1 GB version of the TPC-H [27] data suite was used for the experiments. Shore was configured to use a 500MB buffer pool and a page size of 16KB. The query used is a selection query on the Lineitem table. The selection predicate selects tuples that have been shipped in a 10 day interval (the predicate is on the l_shipdate attribute and the selectivity is around 0.5%). An unclustered index was built on the l_shipdate using a SHORE B-Tree. The unclustered index scan plan sorts the RIDs before fetching them. Table 1 illustrates how the performance of the index plan varies as a function of *f*, the fraction of the data pages containing tuples that satisfy the selection predicate that are already in the buffer pool.

f	Index Scan Time (s)
0	18.78
0.3	17.64
0.5	11.92
0.7	10.39
0.8	7.77
0.9	4.73
1.0	0.80

Table 1: Unclustered Index Scan Performance 800 MB Lineitem table

The time required to execute this query using a table scan is 17.52 seconds. This time is not significantly affected by the buffer contents since the predicate selectivity is around 0.5% and most of the pages of the scan have to be read from disk. The index scan plan (when the buffer cache is empty) takes 18.78 seconds. Since this is greater than the time taken to scan the table, a traditional optimizer would choose to ignore the index plan. As shown in Table 1, depending on the value of f, the relative performances between the two query plans can be quite substantial. In fact, when more than 50% of the required pages are in the buffer pool, the optimizer should definitely choose an index plan. This illustrates how knowledge of the contents of the buffer pool can provide a better query plan. Table 1 illustrated the performance of an index scan plan for an 800 MB version of the Lineitem table, which does not entirely fit in the buffer pool. The same experiment was repeated with a 400 MB version of the Lineitem table in order to examine the trade-off between a sequential scan and an index scan for a table that fits in the buffer pool. Figure 1 graphs the ratio of sequential scan time to index scan time for two different predicate selectivity values (0.5% and 1%). The crossover between the index scan plan and sequential scan occurs at a selectivity of around 0.1%. As a result, a traditional query optimizer would not choose the index scan plan for theses cases. As the graph indicates, a buffer pool aware query optimizer can provide a significant improvement in performance.

Figure 1: Index Scan Performance (400 MB Lineitem table)



To summarize, through the use of both an analytical model and an actual implementation, the results of this section demonstrate that there exist cases in which having knowledge of the buffer pool contents while optimizing queries can effect which plan should be chosen. We have also demonstrated that there are cases where the alternative plans can have significantly different response times. In the next section we consider join queries. Then in Section 4 we describe the architecture of our experimental optimizer and buffer pool.

3. Multi-Table Queries

In this section we examine how the contents of the buffer pool can affect access path selection for joins including both the order in which joins are performed and the choice of join algorithm.

3.1 Join Ordering

In order to evaluate a join between two tables, one of the decisions that the optimizer must make is to select an appropriate join order. Consider a join between two tables A and B and assume hash join is used as the join algorithm. Typically the smaller relation is used to build a hash table in memory and the larger relation is used to probe the hash table [9]. Let the number of pages in the corresponding relations be Na and Nb and that Nb < Na. Assume that the entire table A has been cached in main memory but that the optimizer, unaware of this fact, selects B, the smaller of the two, as the 'build' relation. At run time, B would be scanned and a hash table constructed on it. Then A would be scanned, probing the hash table of B for matches. The scan of B is likely to result in pages of A being ejected from the buffer pool. In the worst case the number of pages that would be read from the disk for the join would be Na+Nb. However, if the optimizer had reversed the join order, it would incur no I/Os while scanning the inner ("A"). Hence, the number of pages read from disk would be Nb. For a cost model that uses only I/O costs, the plan with its join order reversed would be the better plan. In general there is a trade-off between the I/O costs saved (based on the fraction of A that is cached) and the additional CPU costs incurred in hashing the 'build' relation (since it is a larger table).

For example, consider a join between a 400 MB version of the Lineitem table and the Order tables (around 150 MB). Using our experimental prototype with a 500MB buffer pool, the execution times for the two alternative join orderings (using hash join) when the buffer pool is empty are shown below. The "ORDERS join LINEITEM" join which uses the smaller relation as the "build" relation is the better plan.

LINEITEM join ORDERS	13.77 s
ORDERS join LINEITEM	12.90 s

The corresponding times when the Lineitem table is entirely in memory are given below.

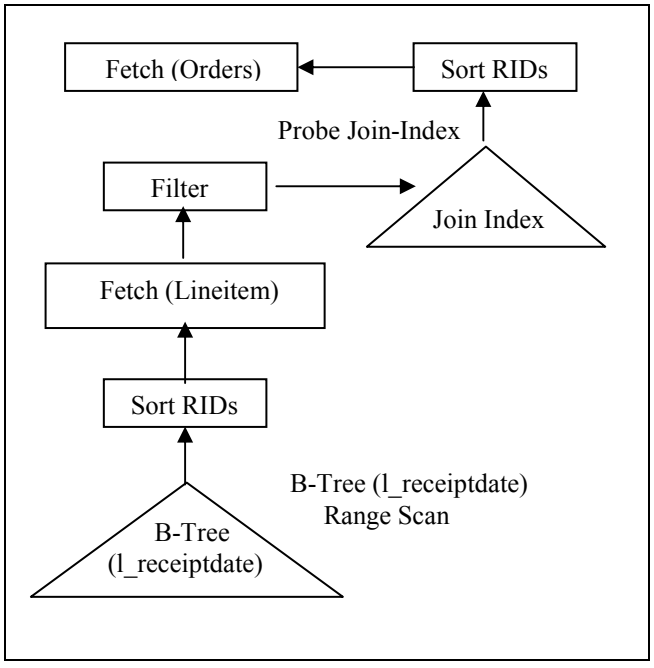
LINEITEM join ORDERS	5.98 s
ORDERS join LINEITEM	12.47 s

The "ORDERS join LINEITEM" join does not exploit the fact that the Lineitem table is cached; as the orders table is scanned, the buffer pool starts replacing pages of the Lineitem table. As the experiment indicates, data caching may have an important effect on join ordering.

3.2 Choice of Join Algorithm

Today relational products employ a wide range of join algorithms such as nested loops, indexed-nested loops, sort merge join, and hash-based techniques [9]. In addition, an optimizer can choose to join suitable indexes (like covering indexes) instead of joining the source tables. Index structures like join indexes [23] also can be used to evaluate joins. A join index between two tables A and B essentially pre-computes the join between the two tables, and stores the mapping between the pair of qualifying RIDs as a pair of B-trees. Using a join index it is possible to evaluate a join between the two tables by probing the index. An interesting point to note is that the trade-off between using join indexes and sequential algorithms (like hash join) for evaluating joins is, in many ways, similar to the trade-off between using an unclustered index scan and a table scan for a single table query. Join indexes will prove to be better than sequential access algorithms until some threshold value in predicate selectivity of the "inner" is reached that would limit the number of probes on the join index. As in the case of unclustered indexes, sorting the RIDs before fetching the corresponding tuples is a standard technique for improving performance.

Figure 2: JINDEX plan



Consider a join query between Lineitem and Orders table (similar to TPC-H Query 12). The Lineitem table has predicates defined on a few columns including the l_receiptdate field. Query 12 requires 4 attributes from the

Lineitem table and 2 attributes from the Orders table. Assume that the following indexes are available.

- An unclustered index on the l_receiptdate field of the Lineitem table.
- A join index between the Lineitem and Orders table on the l_orderkey and o_orderkey attribute.
- Covering Indexes Cov1, Cov2 that include only the attributes that are needed by this query from the Lineitem and Orders tables.

With these alternatives there are a number of possible plans for evaluating the join. The optimizer could use a sequential algorithm (like a hash join) to join the source tables or the covering indexes. Alternatively, the optimizer could decide to use the join index to evaluate the join. As illustrated in Figure 2, the plan in this case would begin with an unclustered index scan on the l_receiptdate field to obtain an initial list of RIDs. Those tuples that satisfy the remaining predicates on the Lineitem table are then used to probe the join index to obtain the RIDs of the qualifying tuples in the Orders table. These tuples are then fetched. The RIDs gathered from both indexes are sorted before fetching the corresponding tuples.

The different plans were executed using the 1 GB version of the TPC-H suite. The Lineitem table is around 800 MB and the Orders table is around 150 MB. A 500 MB buffer pool size was used. With an empty buffer pool the performance of these plans is shown in Table 2. Hybrid hash join was used as the join algorithm. These results indicate that, when available, a traditional optimizer would likely choose the plan that uses both covering indexes.

Query Evaluation Plan	Execution Time (s)
LINEITEM join ORDERS	28.30
LINEITEM join COV2	24.68
COV1 join ORDERS	20.45
COV1 join COV2	16.53
JINDEX plan	21.56

Table 2: Alternate Query Plans

Table 3 presents the response time of the JINDEX plan as a function of f1 and f2, the fraction of the required pages from Lineitem and Orders tables, respectively, already resident in the buffer pool. As these results indicate the trade-off between two alternative choices to the optimizer in this example, the JINDEX and the (COV1 join COV2 plan) could change drastically and the optimizer has the potential to miss better plans.

f1	*f2*	JINDEX plan (s)
0	0	21.56
0.4	0	18.07
0.4	0.4	17.47
0.6	0	14.16
0.6	0.6	13.87
0.8	0	10.62
0.8	0.8	8.58
1	0	8.34
1	1	0.93

Table 3: Performance of JINDEX plan

A similar trade-off would exist for other combinations of algorithms for join evaluation like merge-join and indexed-nested loops. These results, along with the experimental results presented in Section 2, demonstrate that the performance of certain query execution plans (especially those involving random accesses) can vary dramatically depending on the actual contents of the buffer pool; an optimizer that is aware of the contents can make more informed decisions

4. Buffer Pool Aware Query Optimization

4.1 Introduction

In the two previous sections we demonstrated how data pages cached in the buffer pool can influence many of the choices made during query optimization including index selection, the choice of a join algorithm, and the selection of which table should be the "inner" table for a join. Experimental results indicate that an optimizer that exploits knowledge of the contents of the buffer pool can result in the selection of query plans with significantly better performance. In this section, we outline a generic architecture for query processing that is "buffer-pool aware". In this paper, we intend to focus on single table queries and key-foreign key joins which are an important class of join predicates.

A buffer pool caches data pages. Given a PID (page ID) or a particular RID (record ID), it is possible to determine if the page or record is in the buffer pool. A buffer-pool aware query optimizer needs to be able to estimate what fraction of the pages required by a selection or join operator is resident in the buffer pool. This "estimate" would ideally, be accurate and have low computation overheads.

Current optimizers use a "optimize then execute" paradigm as first proposed by System-R [21]. In this approach, during the query optimization phase, statistical estimates of various parameters (derived from pre-computed structures such as histograms) are used for cost estimation. Notice that such traditional schemes will not be effective in our approach as query optimization

requires information that is only available at runtime. The concept of *index pre-execution* is proposed as a candidate solution in the following section.

4.2 Index Pre-execution

In this section, we describe a new technique termed index pre-execution that can provide accurate estimates on the effects of caching and then examine if the overheads of this technique are within acceptable limits. The proposed solution is based on the assumption that probing indexes could be an effective technique to gather runtime information during query optimization.

Given a selection predicate on a relation, it is necessary to estimate what fraction *f* of the data pages containing tuples that satisfy the predicate are already present in the buffer pool. If a B-tree index is available on the attribute on which the predicate is defined, one way to estimate this parameter is as follows. During query optimization, the appropriate range of the B-tree (corresponding to the predicate restriction) is scanned to produce a list of qualifying RIDs (record IDs). Using a RID, it is possible to verify (by probing the buffer manager) if the corresponding page is cached in memory. Thus, by using a list of RIDs that satisfy a selection predicate, it is possible to accurately estimate the parameter *f*. Thus, "pre-executing" a predicate on a B-Tree index can lead to an improved cost estimate for the select operator during query optimization.

Consider a join predicate between two relations (A and B). In order to use cost formulae that reflect the contents of the buffer pool, it is necessary to estimate what fraction of the data pages of B containing tuples that satisfy the join predicate are already present in the buffer pool (the parameter *f2* in section 3). Assume that the optimizer has already pre-executed a suitable B-Tree index to estimate a list of candidate RIDs that satisfy the selection predicate on table A. The optimizer now needs to find the corresponding set of RIDs of B that qualify the join predicate. This can be obtained using either of the two following techniques.

- If a join index were available on the appropriate attributes of A and B, it is possible to use this list of candidate RIDs to probe the join index to generate a list of RIDs of B that would satisfy the join-predicate.
- If an index were available on the join attribute of B (assume the join predicate is A.a = B.b), we could use the RID list of A to retrieve the appropriate A.a values and use these to probe the index on B.b to retrieve the corresponding RIDs of B.

As, in the previous example, the list of RIDs obtained for table B would result in an accurate estimate of the parameter f2. Thus, "pre-executing" a predicate on an appropriate index can lead to more accurate cost estimates for the join operator during query optimization. Index pre-execution is a simple technique that can provide the optimizer with improved cost estimates (that takes the buffer cache into account) for different operator trees. The next step is to examine if the "overhead" of the technique can be made sufficiently low.

4.3 Experimental Evaluation

In this section, we evaluate the performance of index pre-execution for selection and join predicates on our experimental prototype. The buffer pool manager of SHORE was extended to provide the following interface

bool isCached (RID rid)

This function would return true if the page corresponding to the input RID was currently cached in the buffer pool.

Selection Predicates: For selection predicates on a single table, index pre-execution involves the following steps. The relevant predicate is evaluated only on the index pages of a B-Tree index to return a set of RIDs that satisfy the predicate. The RIDs are sorted and "duplicate" (i.e, belonging to the same page) RIDs are eliminated. The fraction of these pages that are present in the buffer pool can now be calculated by using the *isCached( )* function.

The following experiment uses an 800 MB Lineitem table. An unclustered index was built on the l_shipdate column. The following table lists the overhead of index pre-execution (both cold and warm numbers) as a function of the number of RIDs covered by the selection predicate.

RID count	Cold Time (s)	Warm Time (s)
1,000	0.11	0.02
5,000	0.38	0.10
10,000	0.74	0.21
25,000	1.66	0.55
50,000	3.13	1.14

Table 4: Index Pre-execution for selection predicates

The results indicate that index pre-execution for a selection predicate could be effective either when the selectivity is low or if the index were memory resident. But the overhead can be sizable if this were not the case. For instance the time taken to scan the Lineitem table is around 17.5 seconds; index pre-execution for 50,000 RIDs (around 1% predicate selectivity) could hence incur an overhead of nearly 18% which would be prohibitive.

Join Predicates: A buffer-pool aware optimizer must be able to calculate what fraction of data pages containing tuples that satisfy a join predicate is currently cached in the buffer pool. As mentioned in the previous section,

there are two alternative approaches; one that uses a join index and another that uses a key on the join attribute. We evaluate the performance of both these techniques. In this paper we intend to concentrate on key-foreign key joins. The following experiments consider a join between the Lineitem table and the Orders table. The indexes involved in this experiment include a B-Tree index on the key value of the Orders table and a join index between the two tables in addition to the unclustered index on the l_shipdate column of the Lineitem table.

Index pre-execution with a join index proceeds as follows. A range predicate is evaluated on the unclustered index (on the l_shipdate column) to produce a set of candidate RIDs of the Lineitem table. These RIDs are, in turn, used to probe the join index between the Lineitem and Orders table to generate the corresponding RIDs of the Orders table. Note that since the join is between a foreign key and its associated key, each RID of the Lineitem table would find exactly one match in the join index. The fraction of pages is computed from the list of RIDs as described previously. The following table lists the overhead of using a join index as a function of number of RIDs.

RID count	Cold Time (s)	Warm Time (s)
100	0.57	0.012
250	1.28	0.025
500	2.52	0.042
1000	4.67	0.087

Table 5: Pre-execution using join index

Index pre-execution using an index on the join-attribute (in this case the key value of the Orders table) proceeds as follows. A range predicate is evaluated on a B-Tree index to produce a set of candidate RIDs of the Lineitem table. The corresponding records are fetched to extract the foreign key value from the tuple, which is, in turn, used to probe the index on the key value of the Orders table to generate the corresponding RIDs of the Orders table. The overhead of this scheme is illustrated in the Table 6.

RID count	Cold Time (s)	Warm Time (s)
100	1.4	0.013
250	3.05	0.029
500	5.4	0.055
1000	9.64	0.11

Table 6: Pre-execution using index on join attribute

As the results in Tables 5 and 6 indicate, the overhead of index pre-execution for join predicates could be prohibitive for even a small number of RIDS. This is mainly due to effect of random I/Os. For each RID from the Lineitem table, the join index technique requires a random I/O to probe the JI and the other scheme requires two random I/Os, one to fetch the foreign key value and another to probe the index on the key value of the Orders table. Unless most of these random I/Os are serviced from the buffer cache (e.g. when the join index is cached in memory), the overheads are likely to be prohibitive.

4.4 Summary

To summarize, while index pre-execution is a technique that can provide accurate estimates for the effects of the buffer pool contents on select and join predicates, its relatively high cost does not make it practical, especially for join predicates. The experimental results presented in Section 2 and 3 indicate that buffer-pool aware optimization yields the best results when a large percentage of the required data pages are in memory. Thus, it is important to make sure that such cases are not missed during query optimization. Sampling [6] is a simple technique that can be used to obtain the "big-picture" efficiently. We next examine how sampling techniques can be used to enable buffer pool aware query optimization.

5 Sampling Techniques

Index pre-execution works by probing relevant indexes to generate a list of candidate RIDs for each table referenced in the query. These lists are used to infer the effects of the contents of buffer-pool on various operator trees. Instead of calculating the entire list of RIDs that satisfy a predicate, the optimizer can calculate a random sample of candidate RIDs. Relatively small sample sizes should suffice to predict important trends (e.g. when most of the required data pages of a relation are memory resident).

One can consider sampling from indexes if such support were built into B-Trees. Olken [19] explains how B-Trees can be extended to support sampling by using the notion of "random-walks" through the index pages. However such functionality is not common (and is not available in SHORE). Moreover, the main problem with index pre-execution was excessive random I/Os. Thus, we intend to pre-compute random samples of tables and store them in main memory in order to eliminate I/O overheads. The key idea is to store a random sample of the tuples (along with their RIDs) and to "pre-execute" predicates on the samples in order to obtain a random sample of RIDs that satisfy the predicate.

Selection Predicates: Consider a query on a table A with a selection predicate. Assume that a random sample of tuples (Sa) from table A have been computed and stored along with their corresponding RIDs. Hence each sample in Sa is of the form (tuple, RID). The optimizer can evaluate the corresponding predicate on Sa and obtain a random sample of RIDs that satisfy the predicate. If a sufficient number of samples satisfy the predicate, the

estimates obtained could be close enough to the actual value that could have been obtained by index pre-execution.

***Join Predicates**:* Consider a foreign key join between two tables A and B, let the join predicate be between the foreign key value in table A and the key value in table B. Assume that a random sample of (A join B) has been precomputed, say Sab and stored along with the corresponding RIDs of the tuples that participate in the join. Thus, each tuple in Sab is of the form (ARID, Atuple, Btuple, BRID).

Consider a join query between A and B (join predicate involving the foreign key value from table A and the key value from table B) that contains an additional selection predicate (*pred1*) on table A. If an index exists on the key value of table B, one of possible join algorithms is an index nested loops join that uses the index on table B. In order to estimate what fraction of the "inner" relation is cached, a buffer-pool aware optimizer needs to compute a random sample of RIDs of table B that would join with tuples of A that satisfy the predicate. The optimizer can compute this by evaluating the predicate (*pred1*) on Sab and projecting the BRID values (stored as part of the join sample).

This, pre-computation schemes can help the optimizer compute RID samples that satisfy certain selection and join predicates. In the following sections, we describe our prototype system and look at some preliminary experiments that quantify the accuracy and the overheads of pre-executing predicates on random samples.

5.1 Prototype Description

In this section, we briefly outline the extensions required to support RID sampling in our experimental prototype.

Pre-computing Samples: Samples for base tables were computed by scanning the corresponding table and using the reservoir sampling algorithm [17]. For foreign key-key joins (A join B), a sample of the join was obtained by evaluating (Sa join B), i.e. by joining a random sample of the foreign-keys with their corresponding key values. The C random() function was used and was seeded using the current clock time. The pre-computed samples were loaded into main memory when the system starts up.

Buffer pool Manager: The buffer manager needs to support the following two operations in order to facilitate buffer pool aware query optimization.

1. Given a RID, is the corresponding page currently resident in the buffer pool? This is implemented using the *isCached( )* function described in Section 4.3. It is important to ensure that is function is free from side-effects, in particular *isCached( )* function should not fetch the corresponding page into memory; this would bias the random sample and render the estimates inaccurate.
2. What fraction of the pages of a particular relation is currently resident in the buffer pool? This value is used to estimate the cost of a table scan. This can be obtained by keeping a simple counter for each table.

Query Optimizer: Opt++[16] was used as the optimizer in our prototype. The version of Opt++ used employs a dynamic-programming strategy like the System-R optimizer. Opt++ provides support for basic relational operators like Scan, Select and Join. To facilitate buffer pool aware query optimization, a new *pre-execute* operator was added to Opt++. Its primary purpose is to execute predicates on the appropriate samples and generate RID samples in order to improve cost estimation. Consider a simple join query between tables A and B; assume there is a predicate defined on table A and that an index exists to evaluate it. Before initializing the search of the plan space, the optimizer would pre-execute the predicate (defined on table A) on the corresponding random samples computed for that table in order to obtain the fraction of the required data pages that are buffer pool resident. This would, in turn, facilitate using detailed cost estimates for the index plan (similar to those outlined in Section 2). The search would then be initialized with the following nodes

- Scan (Table A)
- Scan (Table B)
- Index Scan (Index on A.attr)
- Pre-execute (A.attr, RID list)

The only difference from the traditional case is the introduction of the pre-execute node. In the next iteration, any pre-execute nodes are first expanded. The optimizer would search for any pre-computed join samples, which can use the sample of RIDs available in the current pre-execute node to generate a sample of RIDs for any other table referenced in the query. In this case, it would amount to probing the join sample using the sample RIDs from table A that were generated in the previous iteration to generate a sample of the RIDs of table B that satisfy the join predicate. Since joins between a foreign key and key involve a one-to-one mapping, this operation can be implemented very efficiently. This sample can, in turn, be used to estimate the effect of the contents of the buffer pool on join evaluation. When the remaining nodes (nodes 1 to 3) are then expanded using traditional join enumeration schemes, the improved estimates can be used to make more informed decisions. The pre-execute operator only serves to propagate RIDs as part of the search process in order to enable improved cost estimates for other operators (like select, join). It is not included in

the final query evaluation plan. Thus, the optimizer search strategy can be extended in a simple fashion to enable buffer pool aware query optimization.

5.2 Experimental Evaluation

In this section, we present some experiments that quantify the accuracy and overheads of pre-executing predicates on samples.

Selection Predicates: The selection predicate used is a range predicate on the l_shipdate column (similar to TPC-H Query 6). The predicate ("1994-01-01" <= l_shipdate < "1994-01-11") spans ten days. Different buffer pools contents can be simulated by pre-fetching specific sub-ranges of this predicate into memory. For instance if we pre-fetch tuples in the range (1994-01-01, 1994-01-06), this would have nearly 50% of the tuples cached in memory. A particular run of the experiment, uses 10 such configurations, by pre-fetching the appropriate range of tuples (10% - 100%). For each such configuration, the actual fraction of pages that are cached can be calculated (say *F-Actual*). By evaluating the original predicate on the set of random samples computed for the Lineitem table, one can predict the fraction of pages that are cached (say *F-Estimated*). Table 7 lists the mean of the absolute error between *F-Actual* and *F-Estimated* for different sample sizes. We report two values; MEAN-ALL estimates the absolute error for all the configurations while MEAN-75% estimates the same only for the cases when *F-Actual* is greater than 75%. As seen in Sections 2 and 3, these are the important cases in which a buffer pool aware optimizer can provide as high as an order of magnitude improvement in performance. The results are averaged over 50 runs (each using a different random sample of the same size).

Sample Size	MEAN-ALL	MEAN-75%
6,000	7.04%	4.60%
12,000	5.91%	3.50%
30,000	4.13%	2.57%
60,000	4.06%	2.39%

Table 7: Sampling for Selection Predicates

The numbers indicate that even for a sample size of 30000 (a 0.5% sample) the error in the difference between *F-Actual* and *F-Estimated* is quite small (within 5% of the actual value). In fact for the important cases measured by MEAN-75% (when more than 75% of the required pages are in memory), even a 0.1% sample (6000 samples) could suffice. The overhead of evaluating the predicate on the sample (when the number of samples is 60000) is around 20 ms which makes it a very efficient technique.

Join Predicates: The join predicate tested is a join between the Lineitem table and the Orders table. There is a selection predicate on the Lineitem table which includes a selection predicate on the l_receiptdate column along with another predicate on the l_shipmode field (similar to TPC-H Query 12). A join index was built for this predicate in order to help simulate different buffer pool configurations. A particular run of the experiment, uses 10 configurations as in the previous case by issuing a suitable pre-fetch query that modifies the predicate on l_receiptdate column and uses the join index to fetch the corresponding Orders tuples. For each such configuration, the fraction of pages of the Orders relation that are actually cached can be calculated (*F-Actual*). *F-Estimated* is calculated by applying the predicates on the samples from the Lineitem table to generate a random sample of RIDs and using these to probe the join sample and get the corresponding tuples of the Orders table (and their RIDs).

Sample Size	MEAN-ALL	MEAN-75%
6,000	11.68%	7.98%
12,000	9.29%	5.99%
30,000	5.88%	3.43%
60,000	4.35%	3.09%

Table 8: Sampling for Join Predicates

Table 8 lists the mean absolute error (MEAN-ALL and MEAN-75%) between *F-Estimated* and *F-Actual* as a function of sample size and is averaged over 50 runs. For join predicates, the table shows that a sample of 1% can provide accurate estimates (within 5%). For the more important cases (MEAN-75%), a sample of 0.5% ought to suffice. The overhead of using the 1% sample is again around 20 ms which makes sampling an attractive solution.

Space Overheads: To enable efficient sampling, the optimizer needs to keep the following *permanently* in main memory; a random sample of all the base tables and join samples for all foreign key-key relationships. As mentioned previously we can avoid duplicating the foreign key values in the join samples. The space overhead in caching a 1% sample for the above experiments (that includes a 1% sample on the Lineitem table and a 1% sample of the join) is around 10 MB. In fact, the space overhead in caching a 1% sample for the entire TPC-H database (including 1% samples for all base relations and 1% join samples for all foreign key-key relationships) would only be around 25 MB, which is certainly affordable. The samples can be maintained in the presence of updates to the base data using techniques similar to those outlined in [1].

To summarize, in order to use cost functions that accurately model the contents of the buffer pool, it is necessary to resort to dynamic query optimization. By calculating a random sample of RIDs for each table referenced in a query during query optimization, it is possible to obtain an improved knowledge on the effect of

the contents of the buffer pool on selection and join operators. This would, in turn, facilitate using better cost estimates while evaluating alternate plans. As the results indicate, by pre-computing a small random sample that is kept memory resident (for base tables and key-foreign key joins), the optimizer has the potential to find better plans that can provide an order of magnitude improvement in performance. For instance, for the join query example discussed in Section 3, the speed-up factor can be as high as 18. We believe that these initial results are promising; some possible extensions to our basic framework are discussed in the next section.

5.3 Extensions

This paper currently assumes that queries are executed immediately after optimization; hence the estimates obtained from the samples reflect the runtime conditions accurately. However it is possible that the state of the buffer pool could change before the query starts executing. We intend to study how the optimizer can be made robust to the transient nature of the buffer pool. One possibility is to use to notion of choose plans [9]. Let P1 denote the plan the optimizer would have originally picked and P2 denote the plan a "buffer-aware" optimizer would choose. The execution plan generated by the optimizer would be a Choose (P1, P2). The Choose node would re-evaluate the predicates on the sample and compare with the *F-Estimated* values obtained during query optimization. If these differ considerably then it would execute P1 instead of P2. This would guarantee that a buffer pool aware optimizer would never be worse than a traditional optimizer, which is desirable.

Another possible avenue for future work is to calculate confidence intervals [6] for the estimates obtained using sampling. If the confidence is not above a threshold (e.g. when the number of samples is not enough) the optimizer can avoid changing plans.

In this paper, we looked at single table queries and single join queries. As part of future work, we intend to consider more expressive queries in particular multi-way joins consisting of key-foreign key predicates. We intend to build on the join synopses [1] work which pre-computes samples for such cases. The main difference would be that the synopses would also need to include the corresponding RIDs.

6. Related Work

Storage trends are discussed in [10]. The "five-minute" rule is discussed in [11], which suggests that data pages accessed every five minutes need to be cached in the buffer pool. In certain applications it is possible to assume that main memory is sufficient to hold the entire database, such applications typically use main memory database systems. There are several main memory database products available commercially including TimesTen [26]. [8] provides an excellent overview of the various issues involved in main memory database systems.

A general overview of sampling techniques is available in [6]. Olken [19] examines in detail how sampling can be incorporated in a database system. While the sampling techniques suggested in [19] are used to obtain a random sample of the data values from a B-Tree, we are interested in obtaining a random sample of RID values that satisfy a predicate. Index pre-execution for join predicates can be implemented using join indexes which were first defined in [23].

In this paper, we present a new case for dynamic query optimization [15]. The runtime parameter that needs to be estimated in this case is what fraction of pages required for a selection or join operator is resident in the buffer pool. The importance of buffer effects on query processing has been previously highlighted in [18] where the authors study how the number of buffers allocated to a query can affect its performance. In this paper we study how data previously cached in the buffer pool (as a result of executing other queries) is likely to affect the choice of query plans in an optimizer. As far as we can tell, this paper is the first to propose having a query optimizer examine the contents of the buffer pool while optimizing queries.

7. Conclusions

Since the cost of main memory continues to drop rapidly there is every reason to expect that an increasingly large fraction of a database's frequently used indices and tables will become "permanently" memory resident in the future. Query optimizers, however, typically assume that all data is disk resident.

Simple analytical models were first used to demonstrate that the optimizer could potentially pick the wrong plan for accessing a single table if the contents of the buffer pool are ignored. Using the TPC-H data suite, we experimentally demonstrated that the performance of certain query plans (especially involving random access) could vary dramatically based on the actual contents of the buffer pool; As a result data cached in the buffer pool could affect many of the choices made during query optimization including index selection, join ordering, and join algorithm selection. An optimizer that reflects on the contents of the buffer pool can result in the selection of query plans with significantly better performance. This is especially important in decision support applications where users issue a sequence of queries and interactive response times are crucial.

In this paper, we examined the changes required to make an optimizer 'buffer-pool' aware. The basic idea is to pre-

execute predicates on appropriate samples during query optimization. This would result in improved estimates on the effects of the contents of the buffer pool on select and join operators. Our experimental results indicate that significant performance improvements (as high as an order of magnitude) is achievable.

8. References

[1] S.Acharya et al. Join Synopses for Approximate Query Answering. Proceedings of SIGMOD 1999.

[2] G.Antoshenkov Dynamic Query Optimization in Rdb/VMS. Proceedings of ICDE 1993.

[3] R.Avnur, J.Hellerstein. Eddies:Continuous Query Optimization. Proceedings of SIGMOD 2000.

[4] P.Boncz, M.Kersten. Monet: An Impressionist Sketch of an Advanced Database System. Proceedings of Basque International Workshop on Information Technology 1995.

[5] M.Carey et al. Shoring up Persistent Applications. Proceedings of SIGMOD 1994.

[6] W.Cochran Sampling Techniques. John Wiley & sons Inc. 3rd edition 1977

[7] G.Copeland, S.Khosefian A Decomposition Storage Model. Proceedings of SIGMOD 1985.

[8] H.Garcia-Molina, K.Salem Main Memory Database Systems: An Overview. TKDE 4(6):1992

[9] G.Graefe Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25 (2): 1993

[10] J.Gray, P. Shenoy Rules of Thumb in Data Engineering. Proceedings of ICDE 2000.

[11] J.Gray, G.Graefe The 5-minute rule revisited and other storage rule of thumb ACM Sigmod Record 26(4):1997

[12] L.Hass et al. Extensible Query Processing in Starbust. Proceedings of SIGMOD 1989.

[13] Y.Ioannidis, S.Christodoulakis On the propagation of errors in the size of join results. Proceedings of SIGMOD 1991.

[14] Z.Ives et al. An adaptive query execution system for data integration. Proceedings of SIGMOD 1999.

[15] N.Kabra, D.DeWitt Efficient Mid-Query Re-Optimization of sub-optimal query execution plans. Proceedings of SIGMOD 1998.

[16] N.Kabra, D.DeWitt Opt++: An Object Oriented Implementation for Extensible Query Optimization. VLDB Journal 8(1):1999

[17] D.E.Knuth The Art of Computer Programming (vol II). Addison Wesley Inc. 1969

[18] L.Mackert, G.Lohman R* Optimizer validation and performance evaluation for local queries. Proceedings of SIGMOD 1986.

[19] F.Olken Random Sampling from databases (PhD Thesis) 1993

[20] V.Poosala et al. Improved Histograms for Selectivity Estimation of Range Predicates. Proceedings of SIGMOD 1996.

[21] P.Selinger et al. Access path selection in a relational database system. Proceedings of SIGMOD 1979.

[22] M.Stillger et al. LEO- DB2's Learning optimizer. Proceedings of VLDB 2001.

[23] P.Valduriez Join Indexes. ACM TODS 12(2):1987

[24] E.Wong, K.Youssefi Decomposition- A Strategy for query proceesing. ACM TODS 1(3): 1976

[25] S.B.Yao Approximating block accesses in database organizations. CACM 20(4): 1977

[26] Times Ten White Paper. www.timesten.com

[27] TPC-H benchmark specification. www.tpc.org