A Pay-As-You-Go Framework for Query Execution Feedback

Surajit Chaudhuri
Microsoft Research
One Microsoft Way
Redmond, WA 98052
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
One Microsoft Way
Redmond, WA 98052
viveknar@microsoft.com

Ravi Ramamurthy
Microsoft Research
One Microsoft Way
Redmond, WA 98052
ravirama@microsoft.com

ABSTRACT

Past work has suggested that query execution feedback can be useful in improving the quality of plans by correcting cardinality estimation errors in the query optimizer. The state-of-the-art approach for obtaining execution feedback is "passive" monitoring which records the cardinality of each operator in the execution plan. We observe that there are many cases where even after repeated executions of the same query with use of feedback from passive monitoring, suboptimal choices in the execution plan cannot be corrected. We present a novel "pay-as-you-go" framework in which a query potentially incurs a small overhead on each execution but obtains cardinality information that is not available with passive monitoring alone. Such a framework can significantly extend the reach of query execution feedback in obtaining better plans. We have implemented our techniques in Microsoft SQL Server, and our evaluation on real world and synthetic queries suggests that plan quality can improve significantly compared to passive monitoring even at low overheads.

1. INTRODUCTION

Using feedback from query execution to improve query plans has been proposed e.g., [12], [21] where feedback consists of recording the number of rows produced by each operator in the execution plan. Such a monitoring approach has low overhead since it requires no changes to the physical operators besides counting the number of tuples output by each operator.This feedback is stored in a feedback cache or warehouse, which is consulted by the query optimizer in conjunction with database statistics when optimizing a query. The accurate cardinalities obtained from feedback can help improve the quality of plans chosen by the optimizer. Feedback obtained from one query can be used by the optimizer when optimizing any query. We refer to this method for obtaining feedback from query execution as *passive monitoring*.

Despite the simplicity and low overheads of passive monitoring, a key question that has not been addressed so far is if the execution feedback from a given query is useful in improving the execution plan for a future execution of *that query itself*. After all, it seems natural to expect that the database system should be able to learn more about a query's characteristics from its own execution without having to necessarily rely on execution feedback from other queries to help "accidentally". In this paper, we critically examine this question. One of the key observations of our paper is that relying only on passive monitoring for gathering feedback can cause the query execution plan to remain stuck with a suboptimal plan, regardless of how many times the query executes. The following example illustrates the limitations of passive monitoring in the context of access path selection.

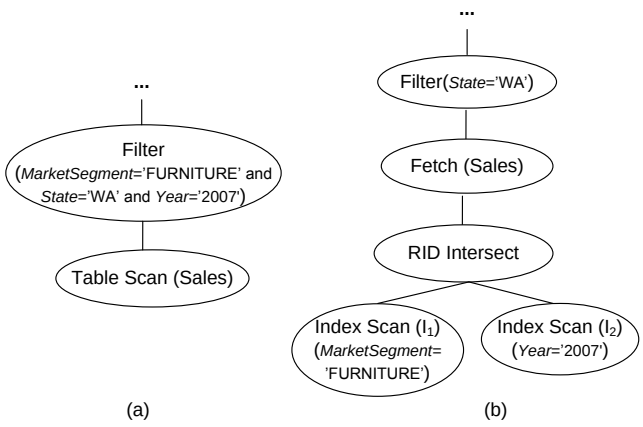

Figure 1. Two alternative plans for a query: (a) Table Scan plan. (b) Index Intersection plan

Example 1: Consider a table ***Sales*** (*SalesId, MarketSegment, State, Year, PaymentType, Amount*) with indexes I_1=(*MarketSegment*) and I_2 = (*Year*), and a query SELECT SUM(Amount) FROM ***Sales*** WHERE *State* = 'WA' and *MarketSegment* = 'FURNITURE' and *Year* = '2007'. Suppose the query optimizer picks a Table Scan plan (see Figure 1(a)), whereas the plan that intersects indexes I_1 and I_2 and fetches the qualifying rows from the table (see Figure 1(b)) is actually lower in execution cost. This can happen for instance, if the predicates *MarketSegment*= FURNITURE' and *Year* = '2007' are negatively correlated. After executing the current plan (a Table Scan), using passive monitoring we only get the cardinality of the expression (*State*='WA' and *MarketSegment*= FURNITURE' and *Year*='2007') from the output of the Filter operator in the plan.

VLDB '08, August 24-30, 2008, Auckland, New Zealand.

However, a cardinality required to accurately cost the index intersection plan in this case is the expression (*MarketSegment*= FURNITURE' and *Year* = '2007'), which cannot be obtained via passive monitoring when executing the Table Scan plan. Thus the optimizer is unable to correct its error using execution feedback obtained by passive monitoring.

As we show later in this paper (Section 2.3), in many cases passive monitoring is also unable to obtain cardinalities of join expressions whose availability can significantly improve plan quality. Thus similar to the access methods problem in Example 1, a plan can also be "stuck" with suboptimal join ordering and join methods despite use of execution feedback. Intuitively, the reason why passive monitoring cannot help correct the optimizer's erroneous choices in many cases is because key cardinality information that is necessary to make the correction cannot be obtained by only examining the output of operator nodes of the current plan. Therefore, even if the database administrator (DBA) or application developer were willing to pay a higher monitoring overhead during query execution it is not possible to obtain additional expression cardinalities if we were to limit ourselves to passive monitoring.

On the other hand, it can be too expensive to collect execution feedback on *all* relevant sub-expressions for a given query that might impact the choice of an execution plan. However, we argue that there are many more opportunities to extend the benefits of execution feedback (without paying excessive overheads) if we are willing to step beyond the confines of passive monitoring. In this paper, we propose a "pay-as-you-go" framework for execution feedback in which a query pays a small additional overhead on each execution so that the plan quality of future executions of the query (or similar queries) is potentially improved. We show that such a framework is able to produce significantly better plans by leveraging execution feedback much more richly while respecting the limits on monitoring overhead defined by the DBA at his/her discretion.

A key enabler of our framework is novel *low overhead* mechanisms for gathering the necessary additional cardinality information from a given query execution plan (Section 3). These mechanisms require modest changes to implementation of existing operators. We refer to these as *proactive monitoring* mechanisms.The second important part of the framework is extending the optimizer to judiciously leverage additional cardinality information obtained through proactive monitoring mechanisms for a query. We refer to this as *plan modification* (Section 4). Plan modification enables generation of a modified plan that ensures collection of the most promising expression cardinalities by piggybacking on the query execution while respecting the DBA specified constraint on the overhead.

We note that the feedback obtained using proactive monitoring can also be used in the same scenarios where feedback from passive monitoring is used today e.g., for refining histograms [1][20], creating statistics [2], during query optimization for improving plans of *other* queries [21] and manual troubleshooting of plan quality.

We have implemented our techniques inside the Microsoft SQL Server engine. Our evaluation (Section 5) shows that: (a) proactive monitoring mechanisms can be utilized with relatively low overhead. (b) The use of proactive monitoring (even with low overhead bounds) results in significant improvement in plan quality in choice of access methods, join ordering and join methods when compared to passive monitoring.

2. ARCHITECTURE

2.1 Assumptions

Queries: Queries can be any SELECT statement with the following restrictions: (a) Selections on a table are a conjunction of predicates. (b) Joins are key-foreign key (K-FK).

Query optimizer: Query optimizers use a cost model to compare different execution plans for a given query. A key input to the cost model is the cardinality of *relevant logical sub-expressions*[1] of the query. The query optimizer considers a set of expressions for a query during optimization. For example, the Microsoft SQL Server optimizer, which is based on the Cascades framework [17], maintains a *memo* data structure. Each node in the memo (group) represents a logical expression. In this paper we assume that the set of relevant expressions for a query is the set of groups in the memo that correspond to relational expressions.

2.2 Query Optimization using Passive Monitoring

As described earlier, passive monitoring functionality in today's DBMSs support the ability to obtain actual cardinalities of operators in the current execution plan. Note that the cardinalities that are available using passive monitoring are dependent on the current plan itself and are often a much smaller subset of all relevant expressions for the query (we discuss this further in Section 2.3). The (*expression*, *cardinality*) pairs obtained from execution can be persisted into a *feedback cache/warehouse* and used as described below. Passive monitoring typically incurs low overhead relative to normal query execution since the additional cost is limited to counting the number of rows output from each operator.

We assume an architecture (e.g., similar to LEO [21]) where the query optimizer can leverage execution feedback to improve accuracy of cardinality estimation. During query optimization, when the optimizer requires the cardinality of a given expression, it looks up the feedback cache (by leveraging existing view matching techniques e.g., [7][16]), and uses the cardinality if available. Otherwise, the optimizer falls back to its default mode of estimating the expression cardinality from the available database statistics. When the query is executed, the cardinalities obtained by monitoring the execution plan are added to the feedback cache. Thus in this architecture, the above cycle of *optimize→execute→monitor* can repeat multiple times for any query, and the execution plan chosen can potentially change based on the feedback information obtained from the set of queries that have executed before it.

In any system that exploits execution feedback for query optimization, there are issues such as maintenance policy for updates, replacement policy for the feedback cache etc. These issues, while important, are orthogonal to the focus of this paper and are not discussed here.

2.3 Limitations of Passive Monitoring

The attractiveness of passive monitoring is that it has low overhead and is easy to implement, since it only pays the

[1] For simplicity, we interchangeably refer to sub-expressions of a query as expressions.

relatively small cost of counting the number of rows output by the operator. However, these very characteristics also limit the scope of passive monitoring. In particular, the cardinality of any relevant expression for the query that does not correspond to an operator in the current plan cannot be obtained via passive monitoring. Thus, even if a DBA were willing to pay a higher overhead (than what passive monitoring incurs), it is not possible to obtain additional cardinalities.

A key factor that determines the effectiveness of execution feedback for improving the plan quality of a query is *which* expression cardinalities are available in the feedback cache for the query optimizer. In Example 1 we presented a case where passive monitoring is unable to obtain an expression cardinality that can overcome suboptimal choice of *access methods*. Below we describe a second example that illustrates that the cardinalities of certain relevant *join expressions* cannot be obtained by today's passive monitoring approach, thus negatively impacting the opportunity to improve the plan.

Example 2. Consider the following query on the TPC-H schema involving the join of tables Customer, Orders, Lineitem and Part:

SELECT * FROM Customer, Orders, Lineitem, Part
WHERE *l_orderkey* = *o_orderkey* and *c_custkey* = *o_custkey* and *l_partkey* = *p_partkey* and *l_shipdate* > '1995-06-01' and *o_orderpriority* = '5-LOW' and *c_mktsegment* = 'MACHINERY'

Note that all joins in this query are K-FK joins. Suppose the current execution plan picked by the query optimizer is the one shown in Figure 2(a). For simplicity we don't show the Filter operators in the figure, and assume that the selection conditions on each table are applied in the Table Scan operators. Consider the case when the predicates *o_orderpriority* = '5-LOW' and *c_mktsegment* = 'MACHINERY' are negatively correlated, thereby resulting in a much smaller cardinality for (Customer ⋈ Orders) than estimated by the optimizer. In this case, the plan shown in Figure 2(b) can be much better since the small cardinality of (Customer ⋈ Orders) allows efficient Index Nested Loops Joins with both Lineitem and Part tables. Observe that using passive monitoring of the current plan (Figure 2(a)), it is not possible to obtain the cardinality of the relevant expression (Customer ⋈ Orders). Thus it is possible that despite repeated executions of the same query, the quality of the plan (in particular the choice of join ordering and join method) may not improve.

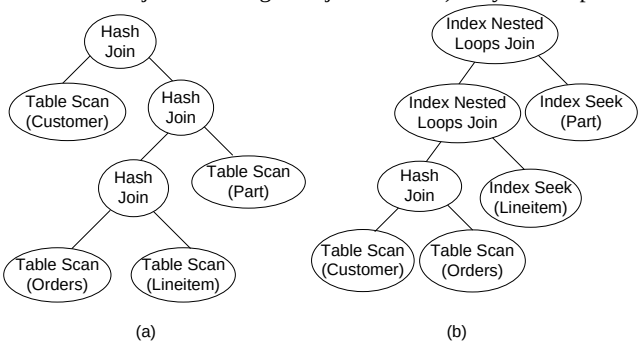

Figure 2. Two different plans for a join query on TPC-H involving Lineitem, Orders, Part and Customer tables.

2.4 A Pay-As-You-Go Framework

We propose a "pay-as-you-go" execution feedback framework in which a query pays a small additional overhead on each execution so that the plan quality of future executions of the query (or similar queries) is potentially improved. In our framework, a DBA can specify a bound on the additional overhead for a query. As with passive monitoring, in our architecture (see Figure 3), the optimizer consults the feedback cache during optimization and uses cardinalities of available relevant expressions to derive a plan P. However, unlike passive monitoring, in this architecture, the optimizer is able to influence *which* cardinalities should be obtained from the current execution of the query by suitably modifying the plan P. This additional step is termed Plan Modification (see Figure 3).

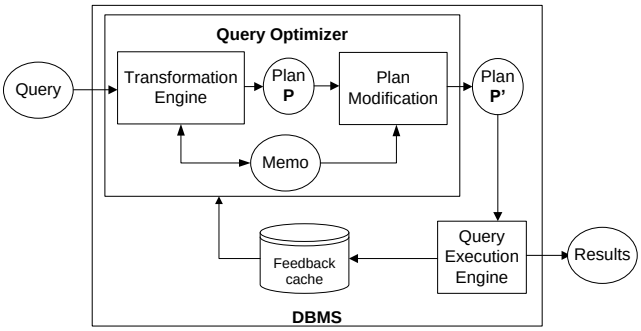

Figure 3. Architecture for query optimization in the presence of proactive monitoring.

They key contributions of this framework include:

(a) A rich set of *proactive monitoring* mechanisms that can be used to obtain additional cardinalities for a range of values of the overhead bound (Section 3). Thus, if a DBA is willing to pay a larger overhead, our mechanisms allow the overhead to be exploited effectively. It is important that these mechanisms are efficient so that they can be leveraged appropriately by a DBA.

(b) A novel *plan modification* step (Section 4) as part of query optimization that modifies the plan P produced by the transformation engine [17] to obtain additional cardinalities using the above proactive monitoring mechanisms. Since plan modification must be able to identify which expressions (among the many relevant ones for the query) are "important" to be obtained, this step is integrated with the optimizer and uses the memo data structure. We note that the plan modification step itself needs to be low overhead since we do not want to significantly increase query optimization time. Thus our algorithms for identifying important expressions as well as changing the current plan use intuitive but lightweight techniques.

In the above framework we model the overhead bound using the optimizer estimated cost, i.e., the optimizer estimated cost of the plan P′ output by plan modification should be no more than ***t%*** higher relative to the optimizer estimated cost of plan P. Of course, optimizer estimated cost may not always accurately reflect execution cost. However, like the case of today's commercial physical design tools (see [22] for an overview); we rely on optimizer estimated cost as a pragmatic alternative to execution cost. We have prototyped the above framework by modifying Microsoft SQL Server.

3. MECHANISMS FOR PROACTIVE MONITORING

In this section, we describe novel mechanisms for obtaining additional expression cardinalities from the current query's execution that cannot be obtained by passive monitoring alone.

Notice that we need mechanisms that can be used for a range of values for the overhead parameter ***t***% (Section 2.4). We present a spectrum of techniques with varying overheads that can be leveraged based on the threshold. These mechanisms involve changes to the server, in particular the predicate evaluator and query operators. We also discuss efficient implementation of these mechanisms in this section.

3.1 Mechanisms for Single-Table Expressions

Using the available indexes effectively is an important responsibility of the query optimizer. In order to determine if an available index (or an intersection of two available indexes) is appropriate to use for the query, the optimizer needs to be able to accurately estimate the cardinality of the predicate (or conjunction of predicates). Such expressions are single-table expressions, i.e., all predicates are on the same table. Thus obtaining accurate cardinality for single-table expressions can be very important for improving suboptimal choice of access methods by the query optimizer. In the discussion below, we assume the current plan can either be a Table Scan or an Index Seek plan and we are given an expression (or set of expressions) for which the cardinality value is required. We use the following query as a running example to illustrate the different mechanisms.

Example 3. **Single table expressions available via passive monitoring.** Consider a query with four predicates on a table: A > 10 and B = 20 and C < 30 and D = 40. Suppose three single column indexes: I_A= (A), I_B = (B) and I_C = (C) exist on the table. If the optimizer chooses a Table Scan operator in the current plan, then only the cardinality corresponding to the expression (A > 10 and B = 20 and C < 30 and D=40) is available via passive monitoring. If an Index Seek on I_A is used to answer the query, then the cardinalities for the expressions (A > 10) as well as (A > 10 and B = 20 and C < 30 and D=40) are available. On the other hand, if an Index Intersection plan of indexes I_A and I_B is used, then the cardinalities for (A>10), (B=20), (A>10 and B=20) as well as (A > 10 and B = 20 and C < 30 and D=40) are available. Note that in the above three plans, cardinalities of expressions such as (C<30) or (A>10 and B=20 and C < 30) etc. are not available from execution.

Our basic mechanisms involve modifications to the predicate evaluator in the database engine. We briefly review the main components of a predicate evaluator. Consider a conjunction of atomic predicates. The predicate evaluator: a) Maintains an *ordered* list of the atomic predicates. b) Typically resorts to *predicate short-circuiting* for efficiency. Thus, in our example, if the predicates are evaluated in the order (A<10 *and* B=20 *and* C<30 *and* D=40) for a particular row, if the predicate (A<10) evaluates to FALSE, then the remaining predicates in the expression are not evaluated. c) Maintains a single counter to count the number of rows that satisfy the predicates. During query evaluation, the predicate evaluator takes as input a tuple and returns TRUE/FALSE. The overheads incurred in predicate evaluation include the cost of evaluating the predicate and the cost of maintaining the counter.

In Sections 3.1.1 to 3.1.3, we use a Table Scan plan to illustrate our mechanisms. We discuss index plans in Section 3.1.4. Finally, we note that for mechanisms presented in Sections 3.1.1 to 3.1.3, sampling techniques can be used to reduce the monitoring overheads. We discuss how sampling techniques can be leveraged in Section 3.1.5.

3.1.1 Prefix Counting

Consider the predicate: (A<10 *and* B =20 *and* C<30 *and* D=40) in Example 3. Let the query plan be a Table Scan operator and assume that the predicate evaluator evaluates the predicate in the left to right order. With passive monitoring we can obtain only the cardinality of (A<10 *and* B =20 *and* C<30 *and* D=40).

Observe that it is possible to obtain the cardinality of each leading prefix of the predicate list with only the small additional overhead of counting. We add one additional counter for each leading prefix whose cardinality we wish to obtain (in this example, (A<10) and (A<10 *and* B=20) etc.). Each time a prefix of the predicates is satisfied for a row we increment the corresponding counter. The limitation of this technique is that it can only obtain a counter if it is a prefix of the predicate list in the Filter operator. For example, it is not possible to obtain the cardinalities for expressions such as (A<10 *and* C<30) or (B=20 *and* C<30) using prefix counting.

3.1.2 Predicate Reordering

Predicate reordering evaluates the predicates in a *different* order than the one chosen by the optimizer. In the example query above, suppose we want to obtain cardinality for the expressions (B=20) and (B=20 *and* C<30). This can be achieved using predicate reordering as follows: the predicates (B=20 *and* C<30 *and* A<10 *and* D=40) are evaluated in the left to right order and prefix counting (Section 3.1.1) is applied. Thus the cardinality of prefix expressions, namely (B>20) and (B>20 *and* C=30) also become available. Observe that predicate reordering is more powerful than prefix counting since it allows obtaining any single expression cardinality by suitably reordering the predicates.

The overhead incurred by predicate reordering is due to the fact that the cost of evaluating the predicates can be higher than if the original ordering was preserved. In the above example, if (A <10) is the most selective predicate and if the predicates (B =20) and (C < 30) are not selective, then the new predicate ordering would not be able to exploit predicate short-circuiting as effectively as the original ordering which used (A < 10) as the first conjunct. Consequently the new reordering would incur more overhead.

3.1.3 Avoiding Predicate Short-Circuiting

As mentioned earlier, the predicate evaluator typically resorts to *predicate short-circuiting* for efficiency. However, if we can modify the predicate evaluator code to bypass the short-circuiting optimization, then a much larger set of expression cardinalities can be obtained. In our running example, assume the order of evaluation of predicates is (A<10 *and* B=20 *and* C<30 *and* D=40). Since (A<10) is always evaluated for every row, the cardinality of that expression can be obtained accurately. If we bypass predicate short-circuiting for the first predicate only, then note that the cardinalities for (A<10) as well as (B=20) become available. This implies that the cardinality of (A<10 *and* B=20) can also be derived. In general, if we bypass predicate short-circuiting for the first *k-1* predicates, the cardinality of *any subset* of the first *k* predicates can be computed. Suppose we need to obtain the cardinalities of both (A<10 *and* C<30) and (B=20 *and* C<30), if we avoid predicate short-circuiting, we can determine the truth value of all the individual predicates (i.e. (A<10), (B=20) and (C<30)) from which the truth value of any subset of predicates can be computed.

Note that the mechanism of avoiding predicate short-circuiting can be used in conjunction with predicate reordering. Consider the

predicate (A<10 *and* B=20 *and* C<30 *and* D=40 *and* E<50) and assume that the original ordering of the predicates is as above, i.e., left to right. If we require the cardinalities (A<10 *and* E<50) and (B <20 *and* E<50), we can reorder and evaluate the predicates using the order (A<10 *and* B =20 *and* E<50 *and* C<30 *and* D=40) and disable predicate short-circuiting for only the first two predicates. In general, to obtain the cardinalities for a given set of expressions, we need to find a reordering with the smallest prefix that covers all the required attributes and disable predicate short-circuiting for the prefix.

Finally, we note that the predicate expression can include expensive predicates (such as those that apply user defined functions). Since we only compute cardinalities of expressions that can affect choice of access methods, we typically avoid short-circuiting for only the simpler predicates.

3.1.4 Index Seek Plans

In Sections 3.1.1 to 3.1.3 we considered mechanisms for proactive monitoring for Table Scan plans. We now consider the case when the plan is an Index Seek plan (the techniques naturally extend to the case of index intersection plans).

Referring to Example 3, let the current plan be an Index Seek on the index I_A (corresponding to the predicate A < 10) where the remaining predicates are evaluated as residual predicates after the tuples are fetched from the table. All the previously described mechanisms (Sections 3.1.1-3.1.3) namely prefix counting, predicate reordering, avoiding predicate short-circuiting are also applicable in the case of Index Seek plans, but in a more limited fashion. Since the tuples fetched from the table are only those that satisfy the predicate (A < 10), we can only get cardinalities of any expressions of the form (A < 10) & (*p*) where *p* is any predicate in the query defined on the columns of the Table T. For instance, if the residual predicates are evaluated in the order (B=20 *and* C<30 *and* D=40), the expression (A < 10 *and* B = 20) can be obtained using prefix counting. However, the cardinality of the expression (B=20 *and* C < 30) cannot be obtained using any of the previously described mechanisms.

Consider the case when the requested expression is (B=20 *and* C<30) (this expression may be relevant because there is an index I_{BC} in the database), then it is possible to get the additional cardinality by using index intersections. For instance, suppose the current plan is Index Seek on I_A; if we modify the plan to an Index Intersection plan between the indexes I_A and I_{BC}, then note that the cardinality of (B=20 *and* C<30) can be obtained when the modified plan is executed. Of course, adding an index intersection incurs an overhead relative to the current plan. The overhead is the cost of scanning the range in the index I_{BC} as well as the cost of intersecting the RIDs satisfying (A<10) with the RIDs satisfying (B=20 *and* C<30). In general, this cost can be non-trivial particularly if the number of rows satisfying (B=20 *and* C<30) is large. However, note that by adding an intersection, the number of fetches from the table cannot increase (and in fact may decrease significantly). Thus index intersection can serve as a useful tool for obtaining additional cardinalities when the original plan is an index seek plan but should be used only when the overhead bound is large enough (see Section 4 for plan modification technique that ensure this).

3.1.5 Reducing Monitoring Overheads using Sampling

The problem of counting the cardinality of an expression such as (A<10 *and* B=20) can be done accurately using *uniform random sampling*. The key idea is that we use the proposed mechanisms (prefix counting, predicate reordering etc.) only for a sample of the input tuples and scale the cardinality obtained using the sample to derive an estimate of the cardinality of the expression.

We note that Bernoulli sampling [13] can be used where each row is given equal likelihood of being chosen independently from any other row. Thus, if the total number of rows in the input to the Filter is N, then the expected number of rows for which predicate short-circuiting is disabled is N.*p* rows. Bernoulli sampling also has the advantage that it does not require us to buffer the rows. Thus, this sampling method incurs no additional memory overhead.

Consider the expressions (B=20) and (B=20 *and* C<30). In Section 3.1.2, we explained how reordering the predicate in the order (B=20 *and* C<30 *and* A<10 *and* D=40) would enable obtaining the cardinality of these expressions. We can leverage sampling for predicate reordering in the following fashion. For a randomly selected fraction *p* of the rows, the predicates are evaluated in the order (B=20 *and* C<30 *and* A<10 *and* D=40). For the remaining (1-*p*) fraction of the rows the predicates are evaluated in the originally chosen order (A<10 *and* B =20 *and* C<30 *and* D=40). By applying prefix counting for the fraction *p* of the rows it is possible to estimate the cardinality of expressions such as (B>20) and (B>20 *and* C=30). This can be implemented by keeping track of both predicate orderings in the evaluator and using Bernoulli sampling to decide which evaluator to use for an input tuple. Similarly, sampling can be used for other mechanisms discussed (such as avoiding predicate short-circuiting) by applying the mechanisms for only a fraction *p* of the input tuples.

The cardinality of an expression *e* can be estimated by scaling the cardinality obtained by the sample. Note that this estimator is an unbiased estimator of the actual cardinality since it is computed on a uniform random sample of the rows. Thus, we can control the overheads of proactive monitoring by using sampling. Another observation is that we only the need the expression cardinalities at the end of execution of the query. Since all rows are considered, the accuracy of the resulting cardinality is not affected by the order in which the rows arrive at the Filter operator. Therefore, this technique is unaffected by how the rows are clustered into pages on disk. In our experiments (see Section 5.1) we find that sampling dramatically reduces the overhead, without significantly affecting the accuracy of cardinality.

3.2 Mechanism for Join Expressions

Similar to the choice of access methods, the choice of appropriate join order and join methods can also have a significant impact on query performance. A common reason why an appropriate join order or method is not chosen is because the cardinality estimation of a join expression is incorrect. Passive monitoring is only able to obtain cardinalities of join expressions corresponding to join operators in the current plan. However, as Example 2 shows, the cardinalities of certain relevant join expressions may not be obtainable by passive monitoring.

Consider the case of a join expression ($R \bowtie_{R.a=S.b} S$) which is *not available* from the current query using passive monitoring. (An example of this is (Orders $\bowtie$ Customer) in Figure 2(a)). In the

discussion below we assume that the selection predicates on both R and S have already been applied. Note that if ($R \bowtie_{R.a=S.b} S$) is an arbitrary join, counting the cardinality of the join requires: (1) Creating a *frequency table*, i.e., count of each distinct value of the join attribute of one relation (say R.*a*) followed by (2) Looking up the above frequency table for each row S.*b* in S. However, for the class of key foreign key joins (which occur commonly in real world queries and are also reflected in benchmarks such as TPC-H) we can implement the above operation more efficiently as outlined below.

We leverage the observation that the *frequency table for the key side relation can be compactly encoded by a bitmap* without loss of information assuming sufficient number of bits since there can be no duplicates in the key column. This makes it possible to monitor the join cardinality efficiently as described below. In this section we describe a mechanism (that we refer to as *bitvector counting*) for obtaining the cardinality of K-FK join expression that is not available via passive monitoring. Observe that for any K-FK join, there is always a well-defined key side relation (say R in our running example) and a foreign-key side relation (S) that can be determined based on the declared key foreign-key relationship in the system catalogs. To apply our mechanism two properties need to hold in the current plan: (1) The key side relation must be scanned prior to the foreign-key side relation in the current plan. (2) Both relations must be fully scanned[2].

For any relation, property (2) holds in the following cases: (a) It is an input to a Hash Join operator in the current plan. (b) It is the outer input to an Index Nested Loops Join. (c) It is an input to a Sort operator feeding into a Merge Join. (d) It is an input to a Merge Join and early termination of scanning of that input does not happen. As we show in our experiments (Section 5.2), for queries against the TPC-H database, property (2) holds quite often.

We maintain a bitmap of size *n* bits initialized to 0 (we discuss below how to determine *n*). Let $\mathcal{H}$ be a hash function that can be applied to the domain of the join columns *R.a* and *S.b* and returns an integer between *1* and *n*. Query execution engines typically already support such hash functions e.g., to implement the Hash Join method. When R is scanned, for each row in R, we set the bit corresponding to $\mathcal{H}$(R.a) to 1. Subsequently in the execution of the plan, when relation S is scanned, we maintain a single counter *c* for tracking the join cardinality. For each row of S, we compute $\mathcal{H}$(S.a) and lookup the bitmap constructed on R. If the bit is set, we increment *c*. At the end of the scan on S, we output the counter *c* as the cardinality of the expression ($R \bowtie_{R.a=S.b} S$).

Example 4. Obtaining key foreign-key join expression cardinalities. Consider the query from Example 2. Suppose the optimizer chooses the plan shown in Figure 4 and we want to compute the cardinality of (Customer ⋈ Orders) from this plan. When the Customers table (the key side) is scanned as part of the build of a Hash Join, for the rows that satisfy the selection on Customers, we construct a bitmap on $\mathcal{H}$ (*c_custkey*). Subsequently, when the Orders table is scanned as part of the build phase of another Hash Join, for the rows that satisfy the selection on Orders, we compute $\mathcal{H}$(*o_custkey*) and lookup the bitmap. If the bit is set to 1, we increment the counter.

Observe that if $n \geq |R|$ (note that R is relation obtained after selections have been applied) and the hash function $\mathcal{H}$ produces no collisions, then *C* is the *exact* cardinality of the join expression. If $n < |R|$, then *C* is an upper bound on the actual join cardinality. If |R| is already known exactly, then *n* can be set to |R| (or higher to reduce chance of collisions). This can happen if the cardinality of R is already available in the feedback cache from a previous execution of the query. Alternatively if cardinality of R is not available in the feedback cache, and R is an expression obtained by applying selections on a base table T (e.g., as in Example 4 on the base table Customers), then *n* can conservatively be set to |T| (i.e., |Customers| in the example). Note that even when |T| = 10 million rows, the memory requirement for the bitmap is modest (only ~1.25MB). In the general case, when R is itself a K-FK join and its cardinality is not available from the feedback cache, we set *n* to the cardinality of the FK side relation. For example, if R = (Customer ⋈ Orders), then we set *n* = |Orders|. As we show in our experiments (Section 5.2.3) bitvector counting is effective even with a modest bitvector size (~1MB),

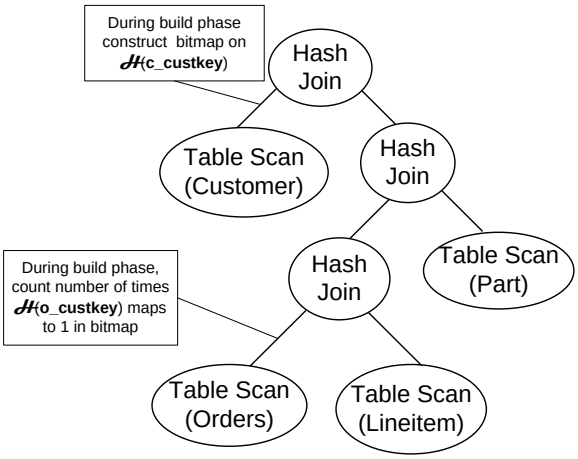

Figure 4. Obtaining a key foreign-key join expression using the bitvector counting mechanism.

Example 5. **Reducing overheads of hashing**. Observe that in Figure 4, since the join column of the expression (Customer ⋈ Orders) *is c_custkey*, which is the same as the join column of the Hash Join where the bitmap is constructed, we can avoid incurring the cost of $\mathcal{H}$(*c_custkey*) since this is already performed as part of the Hash Join. However the cost of $\mathcal{H}$(*o_custkey*) during the scan of Orders cannot be avoided since the join attribute of the current join is different, i.e., *o_orderkey*. Now consider a different execution plan for the same query shown in Figure 5. Suppose we want to obtain the cardinality for the key foreign-key expression (Orders ⋈ Lineitem). This can be achieved as shown in the figure. In this case note that the cost of $\mathcal{H}$(*o_orderkey*) cannot be avoided, but the cost of $\mathcal{H}$(*l_orderkey*) can be avoided during the scan of Lineitem.

Our overhead experiments in Section 5.1 indicate that bitvector counting incurs low overhead (around 2%) compared to normal query execution. We also note that this mechanism can be used to obtain multiple join cardinalities from a given plan as long as the two properties described earlier are satisfied.

[2] Note that the scan of a covering index is also sufficient.

Finally, this mechanism can be viewed as an adaptation of the *bitvector filtering* technique used in the context of parallel database systems (e.g., [14]). The key differences are: (1) Unlike bitvector filtering where the bitmap is always constructed on the join attribute of the *current* join operator, we may need to construct the bitmap on a join attribute for a *different* join expression. For example, in Figure 5 the bitmap is constructed on *o_orderkey* whereas the current join column is *o_custkey*. (2) Bitvector filtering is used to avoid reducing data shipping across nodes in a parallel system, whereas we use the bitvector to count the number of rows in a join expression. (3) Unlike bitvector filtering, as Figures 4 and 5 show, we may "build" a bitvector during a probe phase of a join and "probe" the bitvector during the build phase!

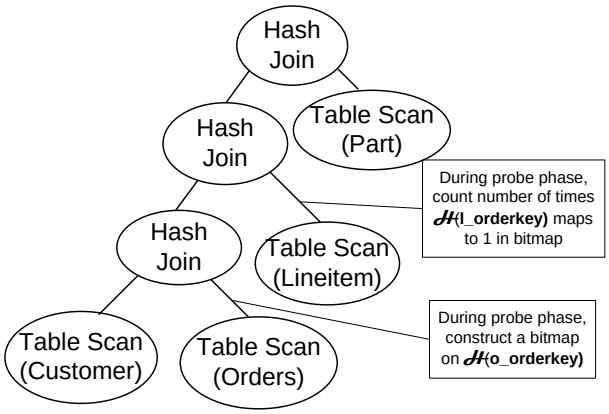

Figure 5. Obtaining (Orders ⋈ Lineitem) from a plan using bitvector counting.

4. PLAN MODIFICATION

In our framework (described in Figure 3) plan modification occurs as part of query optimization. In a typical usage scenario the DBA explicitly specifies an overhead threshold that he/she is willing to tolerate for obtaining additional cardinality information. Recall that the set of relevant expressions is the set of groups that correspond to relational expressions in the memo for which accurate cardinality information has not been obtained. Note that this is more restrictive than the pure syntactic definition of all sub-expression of a query. For example, the memo may not generate cross-products. The optimizer is allowed to modify the current plan P in order to obtain relevant expression cardinalities as long as the optimizer estimated cost of the new plan (P′) does not exceed the cost of the current plan by more than ***t%***.

Plan modification has the following key challenges. First given an expression *e* and the plan P, we need the ability to quantify the cost of modifying P to obtain cardinality of *e*, denoted as **C**(*e*, P). We discuss this in Section 4.1. Second, for a query referencing many tables and predicates, there can be a large number of relevant expressions. Thus the optimizer must be able to differentiate the relative "importance" of these expressions in order to judiciously exploit the available budget for proactive monitoring. We discuss alternative approaches for determining importance of an expression *e* (denoted by **B**(*e*)) in Section 4.2. Finally, we describe the plan modification algorithm in Section 4.3. A key design consideration is that the techniques must be lightweight so as to not increase the query optimization time significantly.

4.1 Costing Plan Modification

We briefly outline how to compute **C**(*e*, P), the cost of modifying the plan P to obtain an expression ***e*.** Our goal is to draw attention to the parameters that affect the cost model. We omit details about the exact functions used and their calibration etc. First if *e* is an expression that corresponds to an existing operator in P, then **C**(*e*, P) is simply the cost of counting the number of rows output by that operator (same as cost of passive monitoring). For single-table expressions (on table T) obtained by using the technique of avoiding predicate short-circuiting (Section 3.1.3), **C**(*e*, P) takes the form ($|T| \times f \times p$), where f is the sampling fraction used (Section 3.1.5), and p is the cost of evaluating the predicate. Note that the optimizer already has cost functions for determining p. For Index Seek plans where the expression is obtained by adding an additional intersection (Section 3.1.4), we use the optimizer's cost model to cost the additional intersection. For a K-FK join expression on (R ⋈ S) that can be obtained using bitvector counting, **C**(*e*, P) is of the form ($|R| \times h + |S| \times l$) where h is the cost of hashing the join attribute and l is the cost of looking up a bitmap. Once again h and l can be obtained by adapting cost functions already present in the query optimizer. Finally, if *e* cannot be obtained from P using any of the above mechanisms, then the cost is ∞.

4.2 Identifying Important Expressions

Intuitively an expression is important if obtaining accurate cardinality for the expression can significantly improve quality of the current plan. However, this definition of importance cannot be directly implemented since it requires obtaining the accurate cardinality for an expression in the first place! Recall that plan modification happens as part of query optimization. Therefore it is critical that the techniques used for identifying importance of expressions are lightweight and simple to implement. We note that the MNSA technique proposed in [10] is too heavy-weight for our purpose, and the sensitivity analysis techniques discussed in [15] is focused only on single-table expressions. Therefore, we focus on low overhead but intuitive measures for ranking expressions by importance. We note however that these approaches do not preclude an offline process which can carefully analyze a pre-specified workload of queries more thoroughly and prune out irrelevant expressions (e.g. using sensitivity analysis). Such an offline process can indeed be useful and can be used to seed a set of "interesting" expression cardinalities to obtain using the mechanisms suggested in this paper. This is an area of future work

As stated previously, we denote the importance of an expression *e* by **B**(*e*). Intuitively, we view the importance measure **B**(*e*) as an indicator of the "benefit" obtained by using the accurate cardinality of that expression. Below we present two alternatives for estimating **B**(*e*). These two techniques are both lightweight and simple to integrate into the optimizer. In our experiments (Section 5), we find that both these techniques are effective in improving plan quality when compared to passive monitoring even when using a small overhead threshold. While these initial results are encouraging, we note that this is an important problem and we expect to further refine these techniques as part of future work.

4.2.1 ASSUM: Number of Assumptions

This measure returns a count of the number of assumptions the optimizer must make in order to estimate the cardinality of the expression. These include independence assumptions among

predicates, containment assumptions for a join, uniformity assumption when interpolating the cardinality within a bucket in the histogram. ASSUM uses a similar intuition to the *nInd* measure used in [8] for choosing among different ways of estimating the cardinality of an expression using existing statistics on views. The rationale is that the optimizer's estimation errors increase with the number of assumptions. Note that this measure can be easily computed for each group as it is derived in the memo [17].

Example 6. We illustrate the ASSUM measure using the example query from Example 2:

SELECT * FROM Customer, Orders, Lineitem, Part
WHERE l_orderkey = o_orderkey and c_custkey = o_custkey and l_partkey = p_partkey and l_shipdate > '1995-06-01' and o_orderpriority = '5-LOW' and c_mktsegment = 'MACHINERY'

Consider the expression e_1 that is the entire query. Each table (except Part) has one selection condition, and hence there is an independence assumption between the selection and the join predicate on that table. Similarly for each of the joins, there is one containment assumption made by the optimizer. Thus there are total of 6 assumptions made by the optimizer. Now consider the sub-expression for this query that is e_2 = (Orders ⋈ Lineitem). For this expression the number of assumptions is 3.

Note that as additional execution feedback becomes available, the number of assumptions for an expression can reduce. For example, if the accurate cardinality of e_2 above is obtained, then the number of assumptions in e_1 = (Customer ⋈ Orders ⋈ Lineitem) is reduced to 2. This is because the optimizer no longer needs to make any assumptions regarding the cardinality of (Orders ⋈ Lineitem). Although the simplest version of the ASSUM measure assigns equal weight to all assumptions, it can be generalized to have different weights for different kinds of assumptions. This allows capturing the relative impact of each kind of assumption on the accuracy of the optimizer's estimate (e.g., independence assumption across predicates typically incurs larger error than a uniformity assumption within a histogram bucket).

4.2.2 SPREAD: Modeling the Estimation Uncertainty

This measure uses the uncertainty in the optimizer's estimate as the basis for deciding the importance of an expression. For each expression we maintain a *lower bound* (LB) and an *upper bound* (UB) of the cardinality of that expression. The measure of importance is the uncertainty of cardinality estimate which we define as *spread* = (UB-LB).

For single-table expressions, the lower bound is initialized to 0 and the upper bound to the cardinality of the table. For a K-FK join expression, the lower bound is 0 and the upper bound is the cardinality of the FK side relation. The key idea is to use feedback obtained from execution to *refine* (i.e., tighten) the upper and lower bounds. We illustrate this idea using the example below.

Example 7. For the query shown in Example 6, consider the expressions e_1 = (Lineitem), e_2 = (Lineitem ⋈ Part) and e_3 = (Lineitem ⋈ Orders). The expressions include all selections on the involved tables. Initially, all of these expressions have LB=0, and UB=|Lineitem|. Suppose after an execution of the query, we obtain the accurate cardinality of (Lineitem), i.e., we know that exactly 10,000 rows satisfy the predicate (*l_shipdate* > '1995-06-01'). This execution feedback allows us to refine the UB of e_2 and e_3 to 10,000 since e_2 and e_3 are K-FK joins where (Lineitem) is the FK side relation. Similarly, it is also possible to refine the LB. For example, if we obtain the accurate value of an expression e_4 = (Lineitem ⋈ Part ⋈ Orders), it can be used as the new LB for the expressions e_2 and e_3.

The LB and UB counters can be maintained in the memo data structure [17] as two new properties of a group. During query optimization, when the optimizer propagates cardinality information (using the feedback cache) among groups, these bounds can be refined using the accurate cardinality information.

4.3 Algorithm

Recall that the plan modification procedure takes as input the set of relevant expressions **E** (whose actual cardinality is not already available) and the current plan **P**. Each expression $\boldsymbol{e} \in \mathbf{E}$ has an associated measure of importance **B**(*e*) (Section 4.2) and **C**(*e*, P) that computes the cost of modifying P to obtain *e*. The goal of the plan modification step is to produce a plan **P′** that obtains a set of expressions $\mathbf{S} \subseteq \mathbf{E}$ such that $Cost(\mathbf{P'})/Cost(\mathbf{P}) \leq (1 + t/100)$, while maximizing $\sum_{e \in S} \mathbf{B}(e)$.

Observe that in our problem, we need to select a subset of the expressions in **E** with maximal $\sum_{e \in S} \mathbf{B}(e)$ such that the set of expressions can be obtained without violating the specified overhead constraint. However, note that for a set of expressions, the **B**(*e*) values may not be independent, and thus not additive. For example, if we can use the current plan to obtain (R ⋈ S), then the additional benefit of obtaining (R ⋈ S ⋈ T) can reduce. However tracking such dependencies across expressions can be non-trivial. For example, when using the SPREAD measure (Section 4.2.2), it is not possible to compute the new benefit without obtaining the accurate LB and UB values. For simplicity, we assume independence between expressions. Notice that the problem is now similar to the 0-1 Knapsack problem where each item has a benefit and cost, and we use the well known and efficient greedy algorithm.

Input: Set of expressions **E.** *Output***:** Modified plan **P′**.

1. **P′** = **P**
2. **For** each $e \in \mathbf{E}$ in decreasing order of **B**(e)/**C**(*e*,**P′**)
3. **If** *e* can be obtained from **P′** by an available monitoring mechanism and obtaining *e* from **P′** does not violate Cost(**P′**) ≤ Cost(**P**)×(1 + ***t* / *100***)
4. Modify **P′** so that expression *e* is obtained when P′ Executes. Update cost of **P′**.
5. **End For**
6. **Return P′**

Figure 6. Procedure for plan modification.

In Figure 6 we describe the procedure for plan modification. Recall that **B**(*e*) is the measure of importance of an expression, and **C**(*e*, **P**) denotes the cost of obtaining expression *e* from plan **P**. Starting with the **P′ = P** (the current plan), we consider adding expressions from **E** in descending order of **B**(*e*)/**C**(*e*, **P′**) (Step 2). Step 3 enforces the necessary checks to see if the expression *e* can in fact be obtained from **P′**. For example, if *e* is a K-FK join expression, we need to check that in **P′** the key side relation is scanned before the FK side relation is scanned (Section 3.2). It also verifies that the overhead constraint is not violated by the addition of *e* to **P′**.

Finally, we note that the above procedure only modifies the current plan P. In general, we can potentially obtain a larger set of relevant expressions if we expand the set of plans considered for modification. If the threshold value t% is high enough we can potentially execute a plan with a different join order if it helps in obtaining important expression cardinalities. For example consider the plans shown in Figure 2(a) and 2(b). These could be alternative plans in the memo. Suppose the first plan is picked as the optimal plan (P). If the second plan is within a cost of ***t***% of P, we can potentially leverage it for plan modification. Naturally, it is important to make sure that the query optimization overheads remain small.

5. EXPERIMENTS

We have implemented the techniques described in this paper inside the Microsoft SQL Server engine. These include the proactive monitoring mechanisms discussed in Section 3 which required changes to the predicate evaluator (for obtaining single-table filter expressions) as well as join operators (for bitvector counting to obtain K-FK join expressions). The plan modification schemes required changes to the query optimizer as described in Section 4.

The goals of the experiments are: a) To quantify the overheads of our proposed proactive monitoring techniques b) To examine the utility of plan modification in real and benchmark datasets. All the experiments were run on a 2.4GHz, 4-processor machine with 4 GB RAM. Numbers we report are based on cold runs so as to eliminate effects of buffering.

5.1 Overheads

There are two sources of overheads in our architecture: (a) Query execution time overhead due to proactive monitoring. (b) Additional overheads in query optimization due to plan modification. We found that (b) was negligible for all queries in our experiments. Thus in this section, we focus on (a). In particular, we study how the overheads of obtaining additional expression cardinalities varies as a function of the number of additional cardinalities obtained. We use a synthetically generated dataset where we can vary the number of predicates and join cardinalities to be obtained in a controlled manner. We generated a synthetic relation (**R**) with 10 million rows and 10 columns (c_1 to c_{10}). Each column has values 1 to 10 million. We used multiple copies of the same relation for the join experiments. We present experimental results using benchmark and real data sets in Sections 5.2 and 5.3.

5.1.1 Mechanisms for Single-Table Expressions

There are two factors that contribute to the overhead of proactive monitoring for single table expressions. The first is the sampling fraction *p*, i.e., the number of rows on which the proactive monitoring mechanisms are evaluated (Section 3.1.5). The second is the number of expression cardinalities obtained using proactive monitoring from a query (*k*). Among the techniques outlined in Section 3, avoiding predicate short-circuiting incurs the largest overhead. We thus report only the overheads of this technique; the results are an upper-bound on the overheads incurred by other techniques outlined in Section 3.1. We generated different queries on table **R** with varying number of predicates on columns c_1 to c_8. In this experiment we study how the overhead of avoiding predicate short-circuiting varies with these two parameters for queries where the original plan is Table Scan.

Figure 7 shows that even when a relatively large sampling fraction like 10% is used, and 8 expression cardinalities are obtained using proactive monitoring from a given query, the average overhead is still no more than around 4%. In fact, for sampling fraction of 1% (used in the experiment in Section 5.2), the overheads are below 1% even when 8 cardinalities are obtained. This expression demonstrates that in practice the overheads of proactive monitoring for obtaining additional single table expression cardinalities can be acceptable.

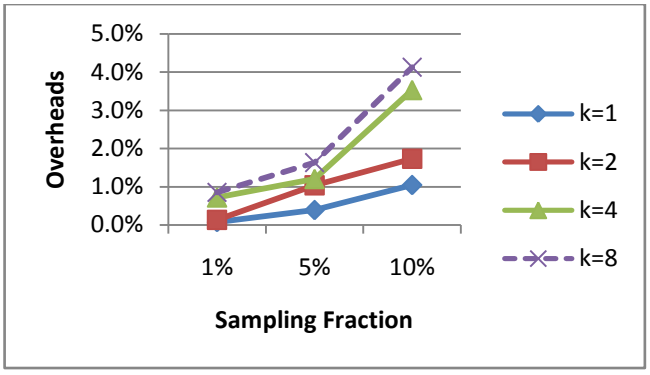

Figure 7. Overhead vs. Sampling Fraction

5.1.2 Mechanisms for Join Expressions

In Section 3.2, we outlined a mechanism for obtaining the cardinality of key foreign-key joins using bitvector counting. In this section, we evaluate how the overhead of this mechanism varies as a function of the number of joins in the query and the number of join expressions obtained using proactive monitoring. We created multiple copies of the table **R** ($\mathbf{R}_1$ to $\mathbf{R}_6$) where relations $\mathbf{R}_2$ to $\mathbf{R}_6$ all have foreign keys referencing different columns of relation $\mathbf{R}_1$. We generated queries of the form ($\mathbf{R}_1 \bowtie \mathbf{R}_2 \ldots \bowtie \mathbf{R}_k$) and forced the join order such that relation $\mathbf{R}_1$ was scanned before any other relation in the plan. The parameter *k* itself was varied from 3 to 6. We measured the overhead of obtaining the cardinality of the join expressions ($\mathbf{R}_2 \bowtie \mathbf{R}_1$), ($\mathbf{R}_3 \bowtie \mathbf{R}_1$), ($\mathbf{R}_k \bowtie \mathbf{R}_1$) etc. using bitvector counting for different values of *k*. In order to measure worst-case overheads, we took care to ensure that the special cases as discussed in Example 5 did not apply for these queries.

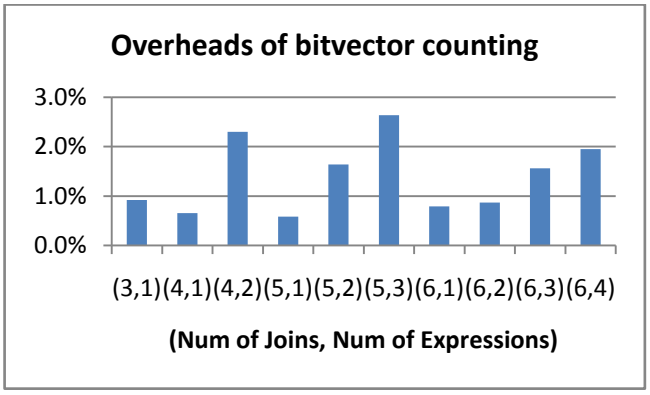

Figure 8. Overheads of Bitvector Counting.

Figure 8 shows the overheads of using bitvector counting as a function of the pair (number of tables joined in the query, the number of expressions obtained). The results indicate that

bitvector counting can be used to obtain as many as 4 additional join expression cardinalities at reasonable overheads (~ 2%).

5.2 Impact on Plan Quality

There are two key aspects to the pay-as-you-go framework discussed in Section 4: a) the module that evaluates the importance of expressions and b) the plan modification component that modifies the existing plan to obtain the desired expression cardinalities. In this section, we fix the algorithm for (a) to ASSUM (Section 4.2.1) and we evaluate the impact of additional counters obtained by plan modification using appropriate proactive monitoring mechanisms. We discuss the tradeoffs between the ASSUM and SPREAD (Section 4.2.2) algorithms to identify important expressions in Section 5.3. We use a sampling fraction of 1% for the single-table mechanisms (Section 3.1.5) for all experiments.

5.2.1 Experiments on Real World Queries

We first present experimental results from a real world Sales database application. The database size is around 1 GB and the query workload consisted of 30 queries, each query consisting of between 5 to 8 joins. In this experiment, we compare PASSIVE with PROACTIVE (with the threshold value *t* set to 1%). We use the ASSUM algorithm to select the expressions. We set the number of iterations to 2 i.e. we run the workload, gather feedback information using the appropriate technique and then rerun the same workload and measure the improvement in the execution times of the query plans. Figure 9 shows the improvement in execution time for both PASSIVE and PROACTIVE-1% when compared to the original plan. We present the results after sorting queries by the improvement of PASSIVE. The first point to note is that PASSIVE is sufficient to correct suboptimal plan choices for 8 of the queries, while PROACTIVE can correct the suboptimal plan choice for 9 additional queries and thus significantly extend the reach of query execution feedback. The regressions (where the improvement is < 0%) indicate the cases where optimizer can potentially pick a "worse" plan even when more accurate cardinalities are available (e.g., due to inaccuracies in the cost model itself).

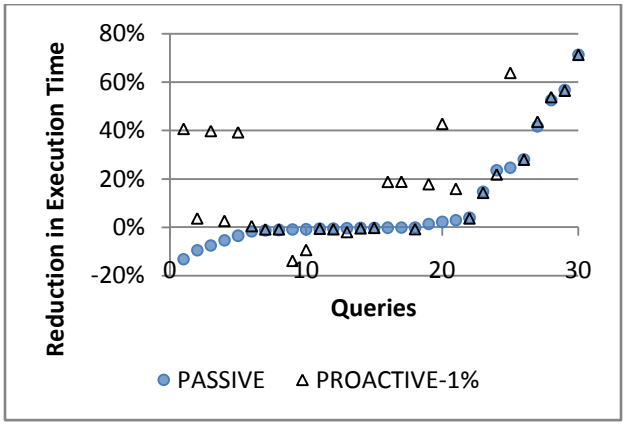

Figure 9. Passive vs. Proactive on Real World Queries

5.2.2 Experiments on TPC-H

We used the modified TPC-H data generator [9] to generate *skewed* data for each column independently with a Zipfian distribution with a skew factor of z=1. Note also that the TPC-H data has limited correlation across tables. For example *o_orderdate* column from the Orders table is correlated with the *l_shipdate* column from the Lineitem table. However, in order to better evaluate the effectiveness of proactive monitoring for join mechanisms we introduced a richer set of correlations across tables. We added correlations between: a) the *c_mktsegment* column in the Customer table and the *o_orderpriority* column in the Orders table (i.e., orders in a certain market segment have the same priority) and b) the *p_mfr* column in the Parts table and *l_discount* column in the Lineitem table (i.e. certain manufacturers offer the same discount rate). We report numbers on both the **1GB** and the **10 GB** version of the TPC-H database.

The query workload was generated by using 2 templates. One is a single-table query on the Lineitem table with predicates on *l_shipdate*, *l_commitdate* and *l_receiptdate* and *l_discount*. The other template is a join of 4 relations, Lineitem, Orders, Customer and Part. The ranges of selection conditions on the columns *l_shipdate*, *l_discount*, *o_orderdate*, *o_orderpriority*, *c_mktsegment* and *p_mfr* were varied in these queries by generating different ranges chosen at random. Indexes were built on all primary keys, join columns and the date columns of the Lineitem table. We used a workload of 100 queries with 50 queries generated using each template and set the number of iterations to 2.

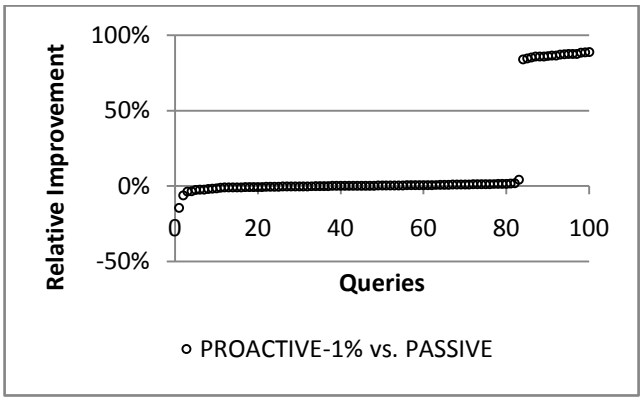

Figure 10. PROACTIVE-1% (TPC-H 1GB)

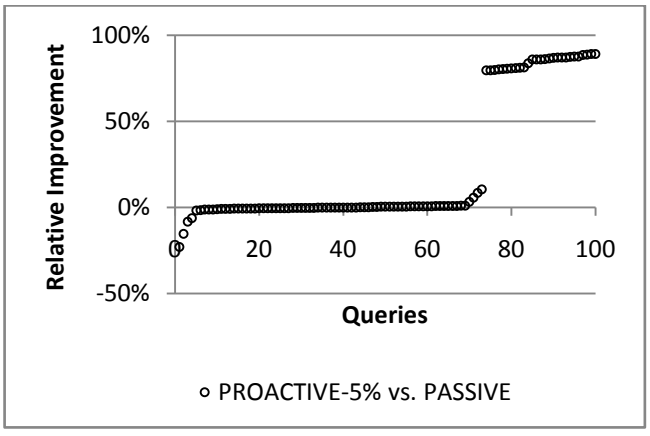

Figure 11. PROACTIVE-5% (TPC-H 1GB)

In Figure 10, we compare the reduction in execution time of the queries using PASSIVE and PROACTIVE (with the overhead threshold value set to 1%). We show the relative improvement of PROACTIVE when compared to PASSIVE. We present the results after sorting queries based on the relative improvement. For instance, Figure 10 indicates that for nearly 20% of the

queries, PROACTIVE-1% is able to provide an additional improvement of nearly 80% when compared to PASSIVE. The results indicate that PROACTIVE has the potential to find much better plans even with small threshold values (1%). The clustering of the results around 80% relative improvement is in contrast to the results in Section 5.2.1. This is because in the TPC-H database, the Lineitem table is the largest table. The biggest savings in performance results when the original plan scans the Lineitem table while the improved plan uses an index instead or uses Lineitem as an inner of an INL Join. For most of the queries, the saving arises due to this fact and hence the relative improvement is clustered around 80%. For a threshold value of 1%, we observed that out of the 18 queries that improved significantly 16 were single table queries from template 1 and 2 were join queries from template 2.

In Figure 11, we show the results for the case of PROACTIVE with the threshold ***t*** value set to 5%. The results indicate that with an increased threshold value, PROACTIVE is able to improve the quality of more plans (in particular PROACTIVE-5% is able to correct suboptimal plan choices in 9 additional join queries than PROACTIVE-1%). The key difference is due to the increased number of join expression cardinalities that can be obtained with a higher threshold value.

We also repeated the above experiments for a 10GB version of the TPCH database and found the results were similar. For instance, Figure 12 shows the results of the experiment for PASSIVE vs. PROACTIVE-1% for the 10GB version. For ***t***=1% we observed the actual execution time overheads were also low. In particular the average was 0.82% and the maximum was 2.1%.

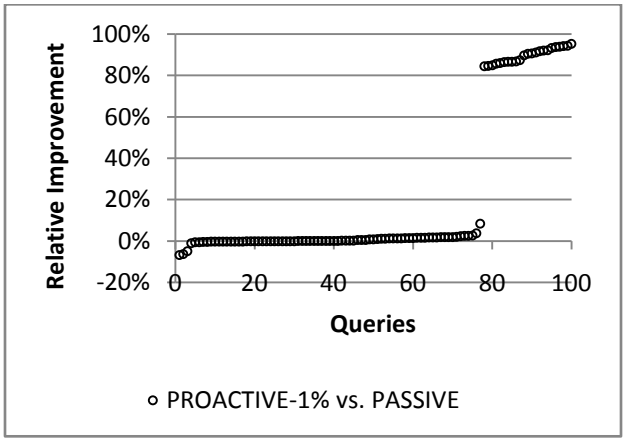

Figure 12. PROACTIVE-1% (TPC-H 10GB)

5.2.3 Sensitivity to Size of Bitvector

As described in Section 3.2, a key parameter that governs the accuracy of the bitvector counting mechanism is the size of the bitvector. In the above experiments we used a default setting of 10MB for the bitvector. To study the sensitivity for the TPC-H 10GB database, we re-ran the join queries from the above experiment where there the plan improved due to availability of additional join cardinalities. We varied the bitvector size as follows: 100KB, 1 MB and 10MB. For a bitvector size of 100KB, we noticed that we obtained the same plan (as with 10MB) for only about 20% of the cases. However, we found that with a bitvector size of 1MB we obtained the same results for all cases as with a size of 10MB. Thus, bitvector counting can be effective even with a modest amount of additional memory. Finally, note that for proactive monitoring mechanisms for single-table expressions there is no additional memory requirement (as discussed in Section 3.1.5).

5.3 Choice of Expressions to Monitor

In Section 4.2, we presented two methods ASSUM and SPREAD for identifying "important" expressions for a query. The experiments of Section 5.2 used the ASSUM method. We repeated the experiments on the TPC-H database using SPREAD instead of ASSUM and found the results were similar. This is because for the queries in the above experiment, most of the relevant expression cardinalities (that can be obtained via proactive monitoring) can be obtained even at a low threshold value such as 1%. Therefore, to better understand the tradeoffs between the two methods, we carried out a controlled experiment in which for each execution of the query, we constrained the execution to obtain *exactly one* expression cardinality as part of feedback. In particular, this is the expression with the highest importance value **B**(*e*) (as determined by that method), among all expression whose cardinality has not yet been obtained. We measured the number of executions (iterations) of the query required to reach the improved plan for each method ASSUM and SPREAD.

We observed that there was no clear winner among the two techniques. Consider the query discussed in Example 2. In this case, there are about 10 expressions in total and ASSUM can obtain the cardinality of the key expression (Customer ⋈ Orders) in 3 iterations while SPREAD can take as long as 7 iterations to get to this expression. This is because SPREAD gives a larger weight to the Lineitem table and if the predicates on the Lineitem table are not selective, SPREAD can potentially exhaust all expressions involving the Lineitem table before choosing (Customer ⋈ Orders). Likewise, consider a case where the important expression required is a sub-expression of a predicate in the Lineitem table. In such cases, SPREAD can converge much faster (in the first few iterations) than ASSUM which would target expressions with a larger number of independence/containment assumptions. We intend to study the tradeoffs between these two techniques in more detail as part of future work.

6. RELATED WORK

Today's commercial DBMSs support the ability to monitor query execution, e.g., the Query Patroller in IBM DB2, Profiler in Microsoft SQL Server, and the Automatic Workload Repository in Oracle. The proactive monitoring mechanisms presented in Section 3 can be viewed as increasing the space of counters that can be obtained using such profiling infrastructure.

The idea of using execution feedback to correct cardinality estimates was introduced in [12]. Previous work in exploiting execution feedback can be classified into two main themes. The first is concerned with using expression cardinalities derived using feedback to correct existing histograms. The notion of self-tuning histograms was introduced in [1], and [20] presents a principled way of using feedback to refine histograms based on the maximum-entropy principle. The second main way of employing feedback is to keep a cache of query expression to cardinality mappings and to utilize it as a supplement to cardinality estimates derived using histograms during query optimization. The LEO project [21] is an example of this approach. The work presented in this paper can be considered to fall in the latter bucket with the key difference being a novel pay-as-you-go framework where the

optimizer proactively monitors the plan at low overhead to extend the scope of execution feedback.

There has been a lot of work on in the area of Dynamic Query Re-Optimization e.g. [5],[18],[19] which uses execution feedback to *dynamically* alter the current execution plan. In contrast to the body of work in dynamic re-optimization, our work is primarily concerned with obtaining cardinalities from the current execution so that future executions of the same (or similar) queries can improve the plans. Interestingly, proactive monitoring can also be leveraged for dynamic re-optimization. Consider the plan shown in Figure 2(a). Observe that if the accurate cardinality of (Customer ⋈ Orders) were available at the end of the build phase on Orders, we could re-optimize the query at that point and potentially switch to plan shown in Figure 2(b) that avoids the scan of the Lineitem table. It is an interesting area of future work to study how proactive monitoring can be leveraged for dynamic re-optimization.

The idea of obtaining relevant expression cardinalities prior to query optimization using single table and join synopses has been studied in [4]. For a query with many relevant expressions the cost of obtaining all cardinalities up-front can be significant. In contrast, our pay-as-you-go approach incurs a bounded overhead on each query execution. The idea of Query-Specific Statistics (JITS) presented in [15] is focused on obtaining statistics of relevant single-table expressions. Our mechanisms can also apply for K-FK joins (Section 3.2). It is interesting to examine if the sensitivity analysis techniques presented in [15] can be extended for the case of joins.

There has been prior work which identifies the need for proactive monitoring. The proactive re-optimization technique presented in [5] also uses sampling to derive cardinality estimates in order to decide whether to re-optimize the query. The difference is that [5] uses sampling to quickly compute estimates that can be obtained by the current execution plan (i.e., the counters that would be obtained by passive monitoring) and does not proactively attempt to obtain other counters. Proactive monitoring has been used in [11] to obtain feedback for helping accurately estimate the *distinct page count*, another important parameter of the cost model used by the optimizer. In contrast, we present comprehensive mechanisms for single-table as well as foreign-key joins that are tailored for obtaining expression cardinalities.

7. CONCLUSION

In this paper, we identify cases where the state-of-the-art approach of passive monitoring for obtaining execution feedback is inadequate for improving choices by the optimizer. We describe a pay-as-you-go framework for execution feedback where each query incurs a small overhead for obtaining additional expression cardinalities. As demonstrated on real and synthetic queries, our low overhead proactive monitoring mechanisms significantly extend the reach of execution feedback for correcting optimizer errors. Identifying other lightweight proactive monitoring mechanisms and exploring alternative techniques for modeling importance of an expression are interesting areas of future work.

8. REFERENCES

[1] A.Aboulnaga, S.Chaudhuri. *Self-Tuning Histograms: Building Histograms Without Looking at Data*. In Proceedings of ACM SIGMOD 1999

[2] A. Aboulnaga, P.Haas, M.Kandil, S.Lightstone, G.Lohman, V.Markl, I.Popivanov,V.Raman. *Automated Statistics Collection in DB2 UDB.* In Proceedings of VLDB 2004.

[3] G.Antoshenkov, *Dynamic Optimization in Rdb/VMS*. In Proceedings of ICDE 1993

[4] B. Babcock and S. Chaudhuri. *Towards a Robust Query Optimizer: A Principled and Practical Approach*. In Proceedings of ACM SIGMOD 2005.

[5] S.Babu, P.Bizarro, D.DeWitt: *Pro-active Reoptimization*. In Proceedings of ACM SIGMOD 2005

[6] S.Babu et al. *Adaptive Ordering of Pipelined Stream Filters*. In Proceedings of ACM SIGMOD 2004

[7] N. Bruno, S. Chaudhuri. *Exploiting Statistics on Query Expressions for Optimization*. In Proceedings of ACM SIGMOD 2002.

[8] N. Bruno, S. Chaudhuri. *Conditional Selectivity for Statistics on Query Expressions.* In Proceedings of ACM SIGMOD 2004.

[9] S.Chaudhuri, V.Narasayya. *Program for TPC-D Data Generation with skew*. ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew

[10] S.Chaudhuri, V.Narasayya. *Automating Statistics Management for Query Optimizers*. In Proceedings of ICDE 2000.

[11] S.Chaudhuri, V.Narasayya, R.Ramamurthy. *Diagnosing Estimation Errors in Page Counts Using Execution Feedback.* In Proceedings of ICDE 2008.

[12] C.M.Chen, N.Roussopoulous, *Adaptive Selectivity Estimation Using Query Feedback*. In Proceedings of ACM SIGMOD 1994.

[13] W.G.Cochran. *Sampling Techniques*. 3rd Edition. Wiley.

[14] D. DeWitt, R.Gerber. *Multiprocessor Hash-based Join Algorithms*. In Proceedings of VLDB 1985.

[15] A.El-Helw, I.F.Ilyas, W.Lau, V.Markl, C.Zuzarte. Collecting and Maintaining Just-in-Time Statistics. In Proceedings of ICDE 2007.

[16] C.A.Galindo-Legaria, M.M.Joshi, F.Waas, M.Wu. *Statistics on Views*. In Proceedings of VLDB 2003.

[17] G. Graefe. *The Cascades framework for query optimization*. Data Engineering Bulletin, 18(3), 1995.

[18] N.Kabra, D.DeWitt, *Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans*. In Proceedings of ACM SIGMOD 1998.

[19] V.Markl et al. *Robust Query Processing through Progressive Optimization.* In Proceedings of ACM SIGMOD 2004.

[20] U.Srivastava et al. *ISOMER: Consistent Histogram Construction Using Query Feedback*. In Proceedings of IEEE ICDE 2006.

[21] M.Stillger, G.Lohman,V.Markl, M.Kandil, *LEO-DB2's Learning Optimizer*. In Proceedings of VLDB 2001.

[22] IEEE Data Engineering Bulleting on Self-Managing Database Systems. Volume 29, Number 3, September 2006.