Diagnosing Estimation Errors in Page Counts Using Execution Feedback

Surajit Chaudhuri, Vivek Narasayya, Ravishankar Ramamurthy

Microsoft Research
One Microsoft Way, Redmond WA USA.
surajitc@microsoft.com
viveknar@microsoft.com
ravirama@microsoft.com

***Abstract*—Errors in estimating page counts can lead to poor choice of access methods and in turn to poor quality plans. Although there is past work in using execution feedback for accurate cardinality estimation, the problem of inaccurate estimation of page counts has not been addressed. In this paper, we present novel mechanisms for diagnosing errors in page count by monitoring query execution at low overhead. Detection of inaccuracy in the optimizer estimates of page count can be leveraged by database administrators to improve plan quality. We have prototyped our techniques in the Microsoft SQL Server engine, and our experiments demonstrate the ability to estimate page counts accurately using execution feedback with low overhead. For queries on several real world databases, we observe significant improvement in plan quality when page counts obtained from execution feedback are used instead of the traditional optimizer estimations.**

I. INTRODUCTION

Cost estimation is central to query optimization. Errors in cost estimation can lead to the choice of a poor execution plan for the query. A key parameter of the cost model is the *distinct page count*, i.e., the number of distinct pages that need to be fetched from a table. This parameter affects the I/O cost estimation of the query and plays a significant role in the choice of access methods (e.g., Index Seek vs. Table Scan) and join methods (e.g., Index Nested Loops (INL) Join vs. Hash Join). Surprisingly, unlike cardinality estimation, there has been relatively little work focused on it. The following example illustrates the importance of the distinct page count parameter.

Example 1: Consider a table *Sales (Id, Shipdate, State, VendorId)*. Assume that the *Sales* table has a clustered index on the Id attribute and there are a total of 10 million rows in the table occupying a total of 200K pages i.e., an average of 50rows/page. Consider a query on a table *Sales* with the predicate: (Shipdate = '06-01-07'). Suppose there is a non-clustered index on (Shipdate) and assume that 50K rows satisfy the given predicate. In order to estimate the I/O cost of the index seek plan that uses the above index, it is necessary for the query optimizer to estimate the number of pages in the Sales table that contain at least one tuple satisfying both predicates. However, this number depends on how the tuples are clustered on disk. For example, if the Sales data is loaded daily basis, then the qualifying tuples could be clustered into very few distinct pages, since the Shipdate is correlated with the Id column generated during data load. In this case, the number of pages could be as small as 1K (=50K/50). On the other hand, if the data is loaded on a per vendor basis, then the Shipdate will not be correlated with the Id column. In this case, the number of distinct pages could be as large as 50K (if each qualifying tuple occurs in a distinct page). Since each page fetch from the table is a random access I/O, the cost of the index seek plan could be dramatically different depending on the clustering effects on disk. Finally, note that the clustering effects on disk can also impact the choice of join method for a query. For example, the *orders* and *lineitem* tables in a sales database may both be clustered by a date attribute. This can affect the cost estimate of the Index Nested Loops join of the two tables.

Today's query optimizers use analytical models based on cardinality (e.g. [10]) to estimate distinct page count. These analytical models typically do not model the on-disk clustering effects and thus the estimates could be highly inaccurate leading to incorrect plan choice. Observe in Example 1 that the error in distinct page count estimation can occur *even when cardinality estimation is accurate*.

In this paper, we adopt the approach of leveraging query execution feedback to compute accurate distinct page count estimates at low overhead. The distinct page counts obtained by monitoring query execution can be used in different ways. For example, the DBA can examine the distinct page count obtained that is relevant for a particular index and compare it with the optimizer estimated value. If the values are significantly different, the DBA can correct the problem using hinting mechanisms to force a better plan (e.g., force an Index Seek plan instead of a Table Scan plan).

The distinct page counts can also be used as part of a feedback based infrastructure (e.g., [17]) that keeps track of estimation errors in the optimizer and leverages it to improve optimization of future queries. While histograms for distinct page counts are typically not supported in today's DBMSs, we note that our mechanisms can potentially be leveraged for the purpose of maintaining such histograms similar to prior work in the area of "self-tuning" histograms [1][16]. We note that using histograms to estimate distinct page counts can require

non-trivial modifications to traditional histograms (see Section VI for more details).

Unlike previous work on using execution feedback for estimating cardinality (e.g., [17]) obtaining distinct page counts poses new challenges. First, we are interested in *distinct* page counts, which is more expensive to compute than counting cardinality since it involves duplicate elimination. Second, the *page id* needed for distinct page counting is only available in the storage engine component of the DBMS. Given the performance sensitive nature of the storage engine and the separation between storage engine and the relational engine component that exists in today's commercial DBMS architectures, judicious choice of techniques and data structures is crucial. Finally, since we leverage execution feedback, our techniques have to work in the context of the *current* execution plan. Note that we may however need to obtain the distinct page counts relevant for costing a plan that uses a *different* access method or join algorithm. For example, there may be an index on a column that is not picked by the optimizer for a query and the current plan is Table Scan. Thus we need to obtain the distinct page count relevant for costing of the Index Seek by monitoring the execution of the Table Scan plan. This introduces additional challenges as described in Section II. A similar problem can arise in join queries as well (Section IV). Thus, obtaining distinct page counts at low overhead from the current plan is a non-trivial problem.

We have built a prototype for computing distinct page counts for access methods and join methods inside the Microsoft SQL Server 2005 engine. Our experiments (Section V) reveal that the overheads imposed on normal query execution are small (typically < 2%). We have also extended the Microsoft SQL Server query optimizer to accept as input the distinct page count for an expression. We evaluate the impact of the page counts obtained on plan quality by injecting the page counts and re-optimizing the query. For several queries on real world as well as synthetic databases, we observe significant improvement in plan quality when the more accurate page counts from execution feedback are used instead of the traditional optimizer estimations.

The rest of the paper is organized as follows. In Section II, we describe the problem of obtaining distinct page counts from query execution. Sections III and IV describe the mechanisms for page counting during query execution for access methods and join methods respectively. We describe our implementation and present an experimental evaluation of our implementation in Section V. We discuss related work in Section VI and conclude in Section VII.

II. DISTINCT PAGE COUNT ESTIMATION USING EXECUTION FEEDBACK

A. Distinct Page Count Parameter

The *distinct page count* parameter represents the number of distinct pages of a table (physically stored as a heap or a clustered index) that need to be fetched by an operator. For the Table Scan operator, this is equal to the number of pages in the table and can be obtained from the catalog. Accurate estimation of this parameter is therefore relevant for costing access methods such as Index Seek and Index Intersection plans as well as the Index Nested Loops join method. Each distinct page involves a new *logical I/O* and if the page is not already present in the buffer pool, it can result in a physical I/O (a random access to disk), thereby significantly impacting the I/O cost.

Note that even though buffering effects could have a significant impact [14] on query cost, most query optimizers today do not model buffering effects. They either consider the buffer to be cold or compute the fraction cached as a function of the number of distinct pages fetched. In either case, our techniques for obtaining accurate distinct page counts can be used to obtain more accurate I/O cost estimates.

The distinct page count parameter is defined with respect to a predicate expression (e.g. *Shipdate='06-01-07')*.

Definition: *Satisfies* (T, PID, *p*): Consider a predicate expression *p* defined on a set of tables including T. For a page PID in the table T, *Satisfies* (T, PID, *p*) is true if and only if there exists a tuple in T belonging to page PID that satisfies *p*. Note that the predicate *p* can include selection predicates as well as join predicates on the table T.

Definition: *DPC* (T, *p*). The distinct page count for a given table T and predicate expression *p* is the count of PIDs in T for which *Satisfies* (T, PID, *p*) is true.

Note that the distinct page count is relevant in the cost model only if there is exists an index that can be used to evaluate the predicate expression *p*. i.e., for costing the Fetch operator following Index Seek, Index Intersection or Index Nested Loops join operators (in this case *p* is the join predicate).

B. Obtaining Distinct Page Counts from Query Execution

As mentioned earlier, obtaining accurate distinct page counts is relevant for costing access methods such as Index Seek and Index Intersection as well as Index Nested Loops Join method. We need to leverage the current execution plan which may be a *different* plan such as Table Scan or a Hash Join in order to obtain the distinct page count. We discuss the issues involved in obtaining distinct page counts from query execution; we first consider access methods and then join methods.

Access methods: Consider the Sales table discussed in Example 1. Consider a query whose predicates are (*Shipdate = '06-01-07' and State = 'CA')*. Suppose there are two non-clustered indexes on (Shipdate, State) and (State) respectively. Thus, the distinct page counts that are relevant are *DPC*(State, *State='CA'*) and *DPC*(Sales, *Shipdate = '06-01-07' and State = 'CA'*). If the current plan involves a Table Scan of the Sales table, then *DPC* (Sales, *State='CA'*) can be obtained from query execution. Intuitively, this is because in a Table Scan operator all pages of the table are scanned, which provides the opportunity to evaluate *Satisfies* (Sales, PID, *State='CA'*) for every page. In fact, it is possible to compute *DPC* (Sales, *p*)

where *p* is any predicate expression, (see Section III-B for details). Thus, given a Table Scan plan, it is possible to detect the accurate distinct page count that is relevant for costing any Index Seek plan.

Suppose the current plan is an Index Seek plan using the non-clustered index (Shipdate, State). Note that we do not have access to *all* the pages for which *Satisfies* (Sales, PID, *State='CA'*) is true. This is because the predicate *State='CA'* is only evaluated for rows that satisfy (*Shipdate='06-01-07'*). Thus we are unable to determine *DPC*(State, *State='CA'*) in this case. Of course, note that *DPC* (T, *(Shipdate = '06-01-07' and State = 'CA')*) can be computed from the above Index Seek plan. Thus it is possible to detect the accurate distinct page count of the current Index Seek plan itself. This can be useful to determine if the Table Scan plan has lower cost than the current Index Seek plan.

Join methods: For join methods, the distinct page count is only relevant if an index exists on the join attribute, in which case it is useful for costing the Index Nested Loops join method as explained in the following example.

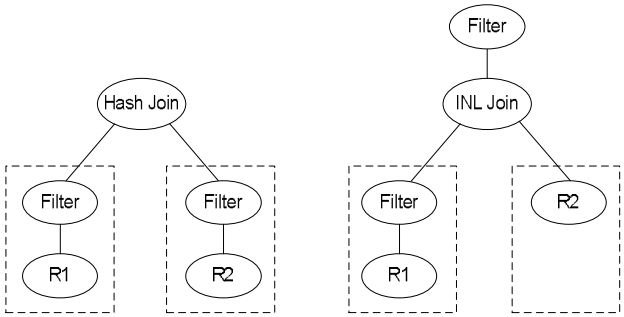

Fig. 1: Choice of Join Algorithm (Hash vs. Index Nested Loops)

Example 2. Consider a join query between two tables R1 and R2. Suppose there is a non-clustered index on the join column of R2. Figure 1 shows two execution plans for evaluating the query. The dotted lines show the operators that execute inside the *storage engine*. This is a common performance optimization technique used by today's commercial database engines. Note that the PID values are available only for operators that execute inside the storage engine. For example in the Hash Join plan, the filters on R1 and R2 are executed inside the respective scan operators for R1 and R2 and the Hash Join operator is executed in the relational engine. In order to accurately estimate the cost of the Index Nested Loops (INL) plan, it is necessary to know the number of distinct pages of R2 that will be fetched by the INL join method. A significant error in the optimizer estimate for this parameter can result in choosing a sub-optimal plan, for example choosing the Hash Join plan when the INL join plan is better.

We therefore need to estimate the number of pages of R2 for which *Satisfies* (R2, PID, *Join-Pred*) is true, where *Join-Pred* is the join predicate between the relations R1 and R2. Note that the current execution can either be Hash Join or INL join. As with the case of access methods, the distinct page count that can be obtained from a plan is a function of the join algorithm used. Consider the case when the plan is Hash Join. The key challenge is the following. Although the predicate *Join-Pred* is evaluated in the Hash Join operator, the PID values of R2 are not available in the relational engine layer. In the scan of R2, even though PID values are available, the predicate *Join-Pred* has not yet been evaluated. We discuss how page counts relevant to join methods can be obtained in Section IV.

C. *Exploiting Distinct Page Counts obtained from Query Execution*

Today's DBMSs already expose modes that output information monitored during query execution. For example, in Microsoft SQL Server, such monitored information can be exposed by Dynamic Management Views (DMVs) or via the *statistics xml* mode [21]. In the *statistics xml* mode the server also returns for each operator in the plan the actual and estimated cardinalities. In our implementation (see Section V for details), we extend the *statistics xml* mode to also output the relevant distinct page counts for access methods and join methods. In this paper, we focus on efficient mechanisms for obtaining distinct page counts. The page counts, once obtained can be leveraged in different ways.

These distinct page counts can first serve as a useful performance debugging tool for DBAs. For example, the DBA can examine the distinct page count that is relevant for a particular index and compare it with the optimizer estimated value. If the values are significantly different, the DBA can correct the problem using hinting mechanisms to force a better plan (e.g., force an Index Seek plan instead of a Table Scan plan). Furthermore, if the optimizer exposes an interface to feedback the accurate page count values to the cost model, the DBA or a client diagnostic/tuning tool can estimate the cost of alternative plans and recommend an appropriate plan hint.

We observe that page counts obtained from query execution can also be integrated into a comprehensive feedback-based infrastructure e.g., [17] that can enable the query optimizer to "learn" about errors in its cost estimates and can correct execution plans. In [17], the feedback information gathered is in the form of (*expression, cardinality*) pairs from the output of each operator in a query execution plan. The framework can be augmented to capture feedback information of the form (*expression, cardinality, distinct page counts*) for appropriate expressions in the query execution plan. Using such a framework would enable reusing the accurate distinct page count for similar queries. Such feedback gathered can also be potentially used to refine histograms for page counts similar to prior work on self-tuning histograms [1][16].

In the following section, we introduce a set of low overhead mechanisms for obtaining distinct page counts that are relevant for costing Index Seek and Index Intersection plans for a particular table in the query. In Section IV we extend these mechanisms for obtaining distinct page count relevant for costing an INL join.

III. OBTAINING DISTINCT PAGE COUNTS FOR SINGLE TABLE ACCESS METHODS

In this section, we present mechanisms for obtaining distinct page counts from query execution. We are given a table T in the query, and a predicate expression *p* on that table, and we want to compute the distinct page count: *DPC* (T, *p*) (see Section II-A). As discussed earlier, this distinct page count is relevant for the query optimizer to accurately cost the I/O of fetching pages from the table that contain one or more rows satisfying the predicate *p*. For simplicity of exposition, we assume that the predicate *p* is a conjunction of atomic predicates.

The current plan could either be an *index plan* or a *scan plan*. Index plans include: (a) Index Seek: Lookup a non-clustered index followed by a Fetch from the table (b) Index Intersection: Lookup two or more non-clustered indexes, intersect the RIDs obtained from each index, followed by a Fetch from the table of the qualifying rows. Scan plans include: (a) Heap Scan (b) Clustered Index Scan (c) Scan of a Covering Index (i.e. an index that includes all the columns in the table that are required by a query).

A scan plan has one additional important property not present in index plans. In a scan plan, all rows in a data page are accessed together, i.e., once all rows in a particular page have been processed, that page is never accessed again. We refer to this as the *grouped page access* property. Note that the grouped page access property does not hold for index plans. This is illustrated in Figure 2.

In Section III-A, we describe a mechanism for obtaining distinct page count for a seek plan. For scan plans, we are able exploit the grouped page access property to provide even more efficient mechanisms for distinct page counting (Section III-B).

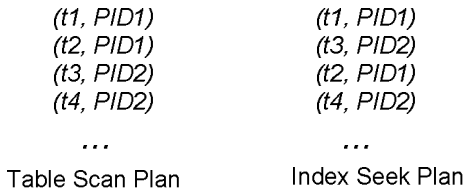

Fig. 2: Order in which pages are accessed.

A. Index Plans

Each time a row is requested by the Fetch operator, the storage engine looks up the page in which that row is located. Since potentially multiple tuples that satisfy the predicate may occur on the same page (i.e., have same PID value), we need to essentially evaluate a COUNT (DISTINCT PID) query over the full sequence of Fetch requests for the seek plan. The exact computation can therefore be expensive and furthermore can consume significant memory. Hence, we use a low overhead alternative as described below.

Probabilistic Counting Approach: We adopt an approach based on probabilistic counting techniques [8][20]. We maintain a bitmap (hashed on the PID value) of the pages that satisfy the predicate. For each row that is fetched from the table, we compute the hash of the PID value for that row and set the corresponding bit in the bitmap. An estimate for the number of distinct PIDs is derived from examining the number of bits set in the bitmap. The algorithm is outlined in Figure 3 and is executed as part of the Fetch operator. We maintain the bitmap in step 3 and scale the estimate based on the number of bits that have not been set (step 6). We assume an *end-of-stream* message is sent when all the input tuples have been consumed. More details of the algorithm are available in [20].

The probabilistic counting approach has desirable accuracy and performance properties. First, it has been shown to be the maximum likelihood estimator [20]. Second, the memory required to ensure high accuracy is very small (typically much less than one bit per page). Thus the main overhead is computing the hash value of the PID and our experiments (Section V) show that this overhead is small relative to normal query execution.

Algorithm DerivePageCount(*p*)
/* *p* is the predicate */
1. Initialize bitmap of size numbits
2. **repeat**
3. If *Satisfies* (T, PID, *p*) is true, compute the hash value using the PID value and set the corresponding bit in the bitmap
4. **until** (End of Stream)
5. Compute the number of bits that have not been set in the bitmap (numzero)
6. **return** $numbits \times -1.0 \times ln\ (numzero / numbits)$

Fig. 3: Probabilistic Counting Algorithm.

We note that another alternative is to use sampling techniques for estimating distinct values e.g., [4]. We can generate a random sample of the *rows* that are fetched from the table using reservoir sampling (e.g., [19]) and apply distinct value estimators for the PID value of the sampled rows. An example of such an estimator is the AE algorithm presented in [4], which computes a set of f_i values (the number of values in the sample that occur exactly i times). Unlike probabilistic counting techniques which examine every row, distinct value estimators based on sampling cannot guarantee high accuracy [4]. We picked probabilistic counting due to its guaranteed accuracy properties as well as the engineering simplicity of incorporation into the storage engine. A thorough empirical evaluation of probabilistic counting vs. distinct value estimation using sampling for the purposes of distinct page counting is part of future work.

B. Scan Plans

For scan plans, i.e., Heap Scan or Clustered Index/Covering Index Scan the grouped page property described earlier holds (see Figure 2). Thus it is guaranteed that if two successive tuples have PID values of PID1 and PID2 respectively (PID1

≠ PID2), then no further tuples in the scan will have the value PID1. As a result there is no need for duplicate elimination of the PIDs. Thus, for scan plans the problem of *distinct* page counting *DPC* (T, *p*) reduces to a simpler problem of *counting* the number of pages satisfying a certain property: in particular, the *Satisfies* (T, PID, *p*) property. This can be implemented as follows. We maintain a counter for *DPC* (T, *p*) that is initialized to 0. For each page PID, as we process the tuples on that page, we check if at least one row belonging to that page satisfies the predicate *p*. If so we increment the counter. At the end of the scan, the value of the counter is the *exact* value of *DPC* (T, p).

If the predicate *p* is indeed evaluated on every row as part of query execution, then the above method is very efficient both in terms of memory required and CPU overhead. It only stores one additional counter and performs a single comparison for each row (to check if the row passed the predicate). However, the current plan may *not* evaluate the predicate *p* on every row. This is due to the use of the well known performance optimization technique of *predicate short-circuiting*. The following example illustrates the problem.

Example 3: Consider the Sales table discussed in Example 1. Consider a query whose predicates are (*Shipdate = '06-01-07' and State = 'CA')*. Suppose the current execution plan is Table Scan and the above predicates are evaluated in the left to right order. Consider the case when a non-clustered index on (State) is present. In order to accurately estimate the cost of a plan that uses the index (State), we need to estimate the number of distinct pages for which Satisfies (Sales, PID, *State = 'CA')* is true, i.e., *DPC* (Sales, *State = 'CA')*. However, the predicate evaluator typically resorts to predicate short-circuiting for efficiency. In this example, if the predicate (*Shipdate = '06-01-07'*) evaluates to FALSE for a row, the remaining predicates are not evaluated. Thus, the predicate *State = 'CA'* may not be evaluated for each row of the table.

Note that in the above example, if the predicate for which the page counts are required was either (*Shipdate = '06-01-07*) or (*Shipdate = '06-01-07' and State = 'CA')*, turning off predicate short-circuiting is not required. For a sequence of conjunctive predicates, there is no need to turn off predicate short-circuiting to obtain the distinct page count corresponding to any prefix of the predicates. However, if the page counts are required for a predicate that is not a prefix of the predicates evaluated, it is necessary to *turn off* the predicate short-circuiting optimization.

Turning off predicate short-circuiting can result in non-trivial overheads. In order to mitigate the overheads incurred by turning off predicate short-circuiting we present an algorithm for distinct page counting based on *page sampling*. We need to estimate the number of distinct pages for which *Satisfies* (T, PID, *p*) is true. Recall that when the grouped page property holds, the problem of distinct page counting reduces to a simpler problem of counting. Since uniform random sampling is known to be an efficient technique for counting, we can estimate *DPC* (T, *p*) accurately by using a *random sample of the pages*. This technique can help reduce the overheads while still ensuring accurate estimation. The algorithm is outlined in Figure 4 and executes as part of the Scan operator.

Observe that the above approach uses Bernoulli sampling to choose a page with probability *f* (step 3). In particular, this implies no additional memory is required. As before a single counter for *DPC* (T, *p*) needs to be maintained. If evaluating the predicate *p* requires turning off predicate short-circuiting, the algorithm does so *only for the tuples* occurring on the pages in the sample. This ensures that the overhead of turning off predicate short-circuiting is bounded. In the above algorithm *f* is the desired sampling fraction. As we show in our experiments (Section V), the overheads of computing distinct page counts using this approach are also small (typically < 2%).

The above algorithm has several desirable properties: (a) It produces an unbiased estimator of *DPC* (T, *p*) (unlike the more general purpose method of probabilistic counting [8][20]) (b) It provides tight error guarantees based on Chernoff bounds. (c) It is lower overhead when compared to the estimators discussed in Section III-A (probabilistic counting and distinct value estimators based on sampling e.g., [4]).

Algorithm DPSample(*f*, *p*)

/* *f* is the sampling fraction, *p* is the predicate */

1. PageCount = 0
2. **repeat**
3. If it is the start of a new PID in the Scan, select the page PID with probability *f*
4. If page PID is chosen as part of the sample, turn off predicate short-circuiting if necessary, and evaluate predicate *p* for all rows in that page.
5. Increment PageCount if at for least one row in PID, *Satisfies* (T, PID, *p*) is true
6. **until** (End of Scan)
7. **return** (PageCount / *f*)

Fig. 4: DPSample Algorithm.

IV. OBTAINING DISTINCT PAGE COUNTS FOR JOIN METHODS

In this section, we describe how to monitor the distinct page count that is relevant for the costing of the Index Nested Loops (INL) join method. Consider the query discussed in Example 2. For the join between the relations R1 and R2 (see Figure 1), the distinct page count for R2 is relevant for choice of the INL join method, when R2 is the inner relation. In particular we need to know *DPC* (R2, *p*) where p is the *join predicate* between R1 and R2. Note that *p* does not include any selection predicates that may be present on R2 since in an INL join method the selection predicate on R2 is evaluated *after* the join.

Of course, the current plan chosen by the optimizer could be any one of: INL Join, Hash Join or Merge Join. Below we

discuss in turn how the above distinct page count can be obtained for each of possible join methods. Similar to the single table case, we note that obtaining distinct page count for a join is relevant only if there exists an index on the join column(s). Note that in a join plan, we can also obtain the page counts relevant for the access methods of the corresponding tables by using the techniques discussed in Section III.

Index Nested Loops Join: Consider the case when the current plan is the INL join plan (see Figure 1). Observe that after looking up the index on the join column of R2, the order in which the rows to be fetched from the R2 table appear is similar to the case of an Index Seek plan (Section III-A). Therefore to obtain the desired distinct page count, the probabilistic counting technique described in Section III-A can be directly applied in this case.

Hash Join: Referring to Example 2, recall that we need to compute *DPC* (R2, *p*) where *p* is the *join predicate*. This can be non-trivial for the following reason. Although the predicate *p* is evaluated in the Hash Join operator, the PID values of R2 are not available there. In the scan of R2, even though PID values are available, the predicate *p* has not yet been evaluated. We handle this problem by exploiting the idea of *bit vector filtering*.

Bit vector filters have been used in the context of parallel database systems e.g. [7] to improve the efficiency of Hash Join. The key idea is to compute a bit vector during the build phase of the join (see Figure 5). For each row in the outer, the value of the join column is hashed and the corresponding bit is set. During the probe phase, for each row of the inner, the value of the join column is hashed, and the bit vector is examined. If the corresponding bit is set, then we know that the page to which that row belongs would be accessed during an INL join. In effect, the bit vector filter can be used as a "derived" semi-join predicate during the probe phase. We can thus invoke the DPSample algorithm (Figure 4) during the Table Scan of the probe table. The modifications required for the Hash Join plan (used in Example 2) are illustrated in Figure 5. Further details of our implementation are described in Section V-A.

If the number of bits used for the bit vector is at least as many as the number of distinct values of the join column of the outer relation, then the above method guarantees the exact page count, since there are no false positives due to collisions. If fewer bits are used, then due to collisions, the page count can be overestimated. In our experiments (Section V), we find small overestimation of distinct page count even when using a relatively small number of bits.

Merge Join: The idea of bit vector filtering for computing page counts described above is also applicable to a Merge Join [9] whenever the bit vector for the outer relation can be computed *before* the inner relation is scanned. For Merge Joins where the outer child is a Sort operator, notice that this property holds. This is because the first GetNext() call to the Sort operator is blocking and returns only after its child is full consumed enabling the construction of the bit vector. Bit vector filtering is also applicable to the case when both inputs are clustered on the respective join columns, i.e., no Sorts on either input. Note that for this case, the *partial* bit vector filter corresponding to the outer rows consumed thus far can be used during the scan of the inner relation to compute the page count. This is because the Merge Join only advances the pointer of the inner relation if the values of the join column match the outer. Note that partial bit vector filters can also be applied for the case when the outer is not sorted and the inner child is sorted.

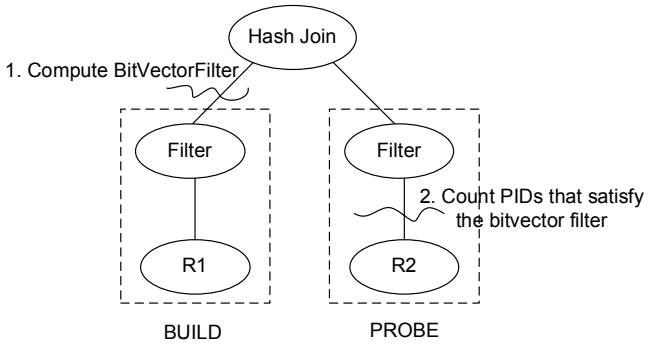

Fig. 5: Distinct Page Counting using Bitvector Filters.

V. IMPLEMENTATION AND EXPERIMENTS

A. Implementation

Functionality: We have prototyped the page counting mechanisms described in Sections III and IV in the engine of Microsoft SQL Server 2005. For a given query, we take as input a set of expressions (e.g., (Shipdate = '*06-01-07*') is an expression) for which distinct page counts are needed. Our functionality is currently enabled in the *statistics xml* mode [21]. In this mode, after the query execution is complete, the server outputs the execution plan in xml format along with additional counters gathered during the query's execution, e.g., cardinality of each operator in the plan. We augment this output with the estimated and the actual distinct page count for each requested expression. We have also implemented a method by which the distinct page count for a given expression can be input to the query optimizer. This allows the user (e.g., a DBA) or a diagnostic/tuning tool to inject a distinct page count value for an expression and obtain the execution plan by optimizing with the injected page count(s). We use this functionality in our experiments to determine the impact of the page counts obtained from query execution on the quality of the plan. Finally, for the case of joins, we have built the mechanisms for INL join and Hash Join.

Handling the RE/SE separation: In Microsoft SQL Server, the PID value for a row is only available in the storage engine (SE). While it is possible to modify the server to expose PID values outside the SE in the relational engine (RE) operators, the engineering complexity as well as performance overheads of such a change can be significant. Since predicate evaluation

of most predicates (except expensive UDFs) is typically done inside the SE, we implemented the necessary changes to turning off predicate short-circuiting in the SE. Information that needs to be passed from SE to RE is accomplished via *callbacks* to the RE layer, for instance for utilizing the bitvector filter for Hash Joins (Section IV).

Impact on cached plans: Our implementation allows the functionality of obtaining distinct page count to be turned on/off without impacting the cached plan for a query. This is because none of our mechanisms requires changes to the plan itself. It allows DBAs to selectively turn on the mechanism when desired without incurring the overhead of recompilation.

B. Experiments

The goals of the experiments are: (a) To demonstrate that distinct page counts can improve the quality of plans significantly, even when the cardinality estimates available are already accurate. (b) To examine the overheads imposed by our mechanisms (Sections III, IV) to obtain distinct page counts. (c) To measure the importance of obtaining accurate distinct page counts in real world databases. Table I lists the set of databases used in our experiments.

TABLE I
DATABASES USED IN EXPERIMENTS

Database	Num Rows (millions)	Num Pages (1000s)	Avg. Num Rows Per Page
Book Retailer	10.8	403	27
Yellow Pages	1	25	39
TPC-H(10GB) Skew factor (Z=1)	60	1121	54
Voter data	4	89	46
Products	0.56	65	9
Synthetic	100	1450	80

Evaluation Methodology: Consider a query Q. Let the current execution plan be *P*. For our experiments, we run the plan *P* and obtain the distinct page counts using the appropriate monitoring mechanisms for the plan. We optimize the query by injecting the distinct page count values obtained from execution feedback. Let the new plan obtained be *P'*. Let the time taken to execute plans *P* and *P'* be T and T'. We report the SpeedUp achieved as: (T – T')/T. In order to isolate the effects of the distinct page counts, we ensured that the plan *P* was generated after injecting accurate cardinality values to the optimizer. All execution times were measured with a cold cache which ensures that effects due to buffering are eliminated.

1) Experiments Using Synthetic Data

We generated a table (see Synthetic database in Table 1) with 100 million tuples having the following schema T (*C1,C2,C3,C4,C5, padding*). The *padding* column is added to make the size of each tuple 100 bytes. *C1* is an identity column with values from 1 to 100000000. Columns *C2* to *C5* represent different permutations of the values in column C1 and are intended to capture different on disk correlations with *C1*. For instance, the column *C2* is equal to column *C1* (representing the fully correlated case), while column *C5* is a random ordering of the column *C1* (and thus is uncorrelated). The intermediate columns represent other data points in between the two extremes. We built a clustered index on column *C1* and non-clustered indexes on columns *C2* to *C5*.

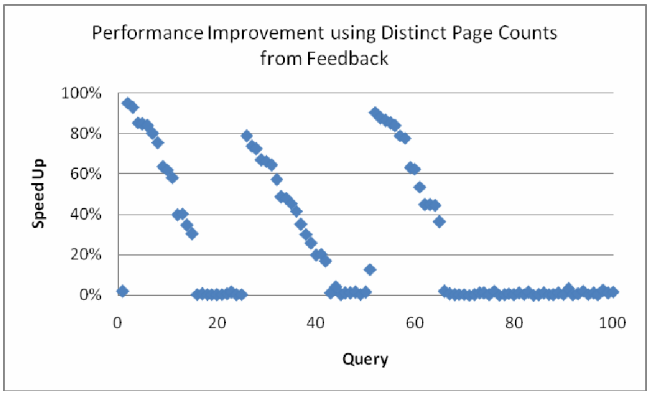

Fig. 6: SpeedUp for single table queries.

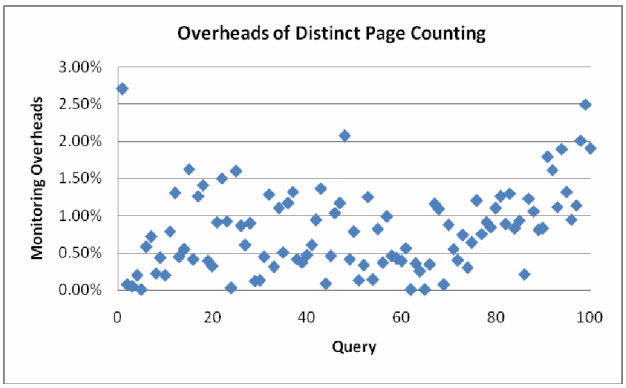

Fig. 7: Overheads for single table queries.

Single Table Queries: We generated queries of the form **select** count(*padding*) **from** *T* **where** *Ci* < val. The selectivity values were chosen randomly from 1% to 10%. We observed for selectivity values greater than 10%, the Table scan plan was optimal irrespective of errors in the optimizer estimate of distinct page counts. We used a query workload of 100 queries (25 queries each for the columns C2,C3,C4,C5). We ensured that the cardinality estimates were accurate by injecting the accurate cardinalities (which were obtained offline). Using distinct page counts obtained from query execution changes the plan in many cases, thereby significantly improving the performance of queries. Figure 6 illustrates the speed up for the query workload, while Figure 7 illustrates the monitoring overheads for the queries. We note that the performance overheads are small (less than 2% for most queries). We observed for most queries that the plan changes from Table Scan to Index Seek. This is because the Microsoft SQL Server optimizer assumes independence between the clustering column and the index column. As a result, the best improvements in Figure 6 are for queries involving columns C2, C3 and C4. For queries with predicates

on column C5 (queries 75 to 100) which is uncorrelated with C1, the distinct page counts do not lead to improvements since the optimizer's estimates of the distinct page count are already quite accurate.

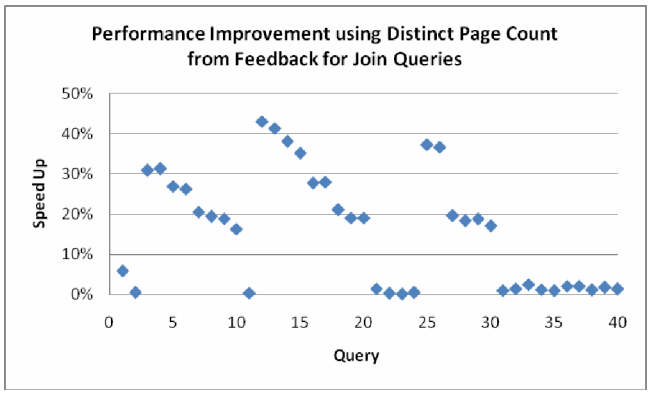

Fig. 8: SpeedUp for join queries.

Join Queries: We created a copy of table T called T1 and created a clustered index on the table column T1.*C1*. We then generate join queries of the form:

select count(*T.padding*) **from** *T, T1*
where *T1.C1* < *val* and *T1.Ci* = *T.Ci*

The optimizer can choose a Hash join for evaluating this query or choose an Index Nested Loops plan using table T as the inner and using the index on Ci column. By varying the Ci column, we can vary the number of pages fetched for a given selectivity value. We used a workload of 40 queries and picked selectivity values for the outer such that the distinct page value could influence the plan choice. For selectivities beyond a threshold value (around 7%), we found that the Hash Join is optimal. Figure 8 illustrates that distinct page counts can provide significant benefits for the case of join queries. In our experiments, the maximum performance overhead due to bitvector filtering that we observed was 2%. Using a bitvector filter of a modest size (less than 1% of the table size) was sufficient to yield high accuracy.

Effectiveness of page sampling: In the previous set of experiments, since the queries have only one predicate there was no need to turn off predicate short-circuiting (Section III-B). In order to understand the performance implications of turning off predicate short-circuiting, we performed the following experiment. We used a set of queries with increasing number of predicates. Figure 9 illustrates how the monitoring overheads for obtaining all the relevant distinct page counts increase with the number of predicates. For this experiment, we used page samples of 1%, 10% and 100% (i.e., full scan). While obtaining the exact distinct page counts is feasible for the Table Scan operator, the overheads of turning off predicate short-circuiting for *all* tuples is clearly impractical. Thus, in order to ensure that the monitoring mechanisms scale to a larger number of predicates, it is *essential* to use the DPSample algorithm. Note that at 1% sampling, the DPSample algorithm incurs an overhead of around 2% and provides an estimate of the distinct page count whose maximum error is 0.5%.

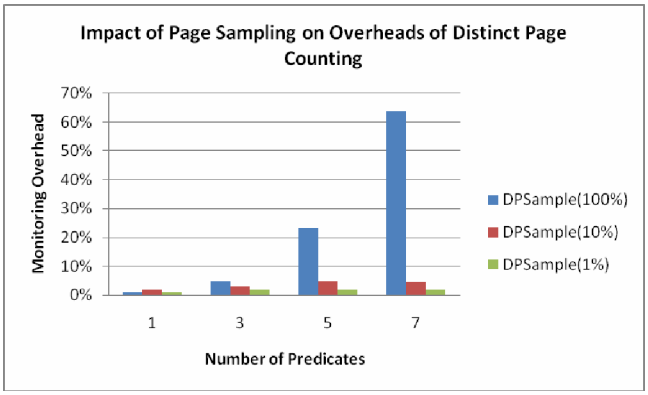

Fig. 9: Effectiveness of Page Sampling.

2) Experiments Using Real World Databases and TPC-H

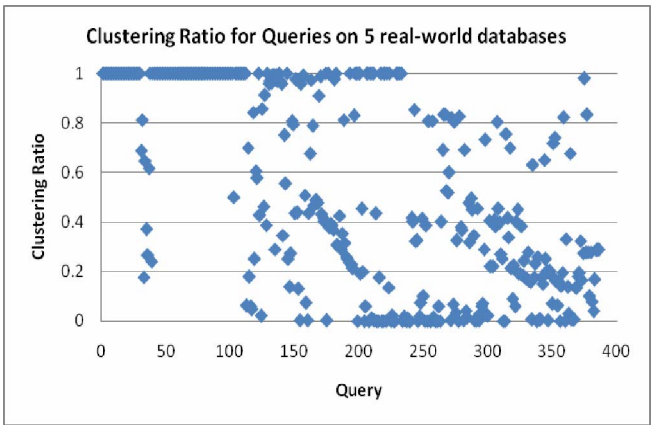

Fig. 10: Page Clustering for Real Datasets.

In this section we measure the importance of obtaining accurate distinct page counts in real world databases. We use the TPC-H database as well as 4 real world databases. The set of databases used and their properties are listed in Table I. Consider a query of the form: SELECT … FROM T WHERE C = Value. Suppose the predicate in the WHERE clause is satisfied by *n* rows. Also, let the number of rows per page be *k*, and the total number of pages in the table be P. Then, the *lower bound* (LB) on the *number of pages* in the table that must be accessed to fetch the *n* rows is *n*/*k*. The *upper bound* (UB) on the number of pages that must be accessed to fetch *n* rows is *min* (*n*, P), since in the worst case, each of the *n* rows may occur in a different page (but not exceeding P in total). The actual number of pages (N) must therefore satisfy: LB ≤ N ≤ UB. Consider the following measure:

Clustering Ratio (CR) = (N-LB) / (UB-LB)

Note that 0 ≤ CR ≤ 1. CR = 0 implies that N = LB (fewest number of pages need to be fetched), which occurs when the values of column C on disk are fully correlated with the table clustering. On the other hand, (CR = 1) => (N = UB), and

therefore the largest number of pages possible for the predicate need to be fetched. Figure 10 shows the Clustering Ratio for several queries on 5 real world databases (see Table 1 for details). Note that since the Clustering Ratio is normalized, we can show all the results in the same plot. We show queries whose selectivity is less than 10%. For different predicates the Clustering Ratio varies widely. The mean Clustering Ratio for the above data points is 0.56 whereas the standard deviation is 0.4! This experiment suggests that simple analytical formulas may be insufficient to capture the clustering effects in real world database.

Impact on plan quality for real-world databases: For this experiment, we generated queries as described above. Figure 11 shows the speed up obtained when we use the distinct page counts obtained from execution feedback for 80 queries for different columns across the real world databases from Table 1. For example, in TPC-H we use the three date columns on the *lineitem* table. The results suggest that our techniques can have significant benefits for real world data sets.

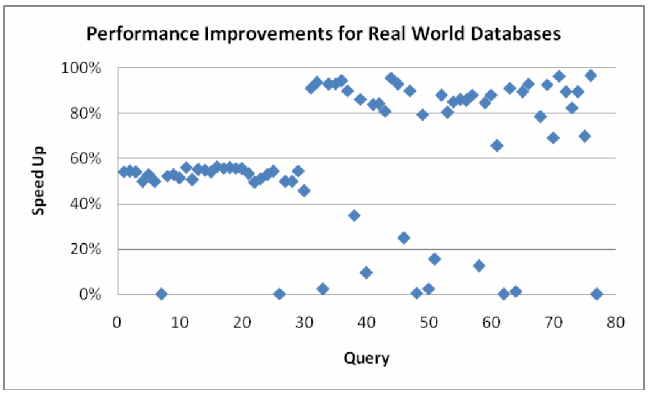

Fig. 11: SpeedUp for Real World Databases.

VI. RELATED WORK

Early work [6][18] showed how on-disk clustering effects can result in inaccurate distinct page count estimation. We can consider using histograms for distinct page count estimation (similar to cardinality estimation). However, this may require non-trivial extensions. For example, unlike traditional histograms the distinct page counts are not additive across buckets since tuples belonging to two different buckets can belong to the same page. We also note that modeling clustering effects for a join method requires the ability to create the histogram of distinct page counts over a join expression (similar to [3]), which is non-trivial. A more detailed examination of how the techniques presented in this paper compare with a histogram-based approach is part of future work. It is also interesting to examine whether synopsis structures such as those presented in [2] can be leveraged for distinct page count estimation.

There has been previous work that focused on correcting errors in cardinality estimates utilizing execution feedback to obtain the cardinalities of query expressions [17]. This paper uses execution feedback to obtain accurate estimates of a different parameter of the cost model, namely distinct page count. A characterization of the sensitivity of query optimization to storage parameters such as disk rates is presented in [15]. Qin et al. [13] present an adaptive approach for calibrating the cost model dynamically based on execution feedback. Our focus in this paper is complementary to this work since we present mechanisms to obtain actual distinct page counts for a query, which could be used in the above framework for calibrating the cost model.

The problem of accurately estimating page fetches is also considered in [14]. Their approach is to use pre-computed samples and execute suitable predicates on these samples to estimate what fraction of the required pages is cached in the buffer pool. In this paper, we do not address the buffering effects and instead focus on distinct page counts. While the buffer pool contents can change (even during the execution of a single query), distinct page counts can potentially be reused to correct estimation errors in future queries having similar predicates. Moreover, for obtaining distinct page counts of join predicates, the pre-computed samples need to be extended to include join synopses, which is non-trivial. Our focus is on a lightweight framework that can be used to detect errors in estimates of the page counts and correct them.

There has been prior work on using sampling for estimating the number of distinct values of a relation [4]. As discussed in Section III-A, these techniques are applicable to our problem. However, when the *grouped page access* property holds, we are able to engineer a more efficient and accurate method that is specific to our problem.

Bit vector filtering has been used to optimize the performance of hash based query operators. Some of its applications include reducing the overhead of communication in parallel database systems [7] and in efficiently processing the recursive partitioning case for hash joins [9]. In this paper, we present a novel application of this technique to help in distinct page counting. We note that unlike the traditional usage, where bitmaps are only maintained for partitions that are spilled to disk, we need to maintain the bitmaps for the entire relation. However, as we demonstrate in our experiments (Section V-B), the overheads of our bit vector filtering implementation are small.

VII. CONCLUSIONS

DBAs today have limited support to diagnose whether a poor plan choice by the optimizer for a given query was due to inaccurate estimation of distinct page counts. In particular, DBMSs do not provide the ability for obtaining accurate distinct page counts. We present low overhead mechanisms that can be used to obtain accurate distinct page counts by leveraging query execution feedback. The distinct page counts thus obtained can be utilized in multiple ways. DBAs and client tools can potentially correct a poor plan choice using plan hinting mechanisms. Distinct page counts obtained can also be used by the optimizer itself to improve the cost estimation for future queries or to update histograms on page counts. A careful study of such feedback driven optimization for page counts is an interesting direction of future work.

ACKNOWLEDGMENT

We would like to thank Raghav Kaushik and the anonymous referees for their feedback on the paper.

REFERENCES

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

[2] K.Beyer et al. On Synopses for Distinct-Value Estimation Under Multiset Operations. In Proceedings of ACM SIGMOD 2007.

[3] N.Bruno, S.Chaudhuri. Efficient Creation of Statistics over Query Expressions. In Proceedings of ICDE 2003.

[4] M.Charikar, S.Chaudhuri, R.Motwani, V.Narasayya. Towards Estimation Error Guarantees for Distinct Values. In Proceedings of PODS 2000.

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

[6] S.Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. ACM TODS, Vol 9, No 2, 1984.

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

[8] P. Flajolet, G.Nigel Martin. Probabilistic Counting Algorithms for Database Applications. J. Comput Syst Sci 31(2): 1985.

[9] G.Graefe. Query Evaluation Techniques for Large Databases.

[10] L.F Mackert, G.M.Lohman. Index Scans using a finite lru buffer: A validated I/O model. ACM Transactions on Database Systems, 14(3):401-424, Sept. 1989.

[11] F.Olken, D.Rotem. Random Sampling from Databases. A Survey. Statistics and Computing. March 1995.Vol 5.

[12] V.Poosala, Y.Ioannidis, P.Haas, E.Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In Proceedings of SIGMOD 1996.

[13] Y.Qin, K.Salem, A.Geol. Towards Adaptive Costing of Database Access Methods.

[14] R.Ramamurthy, D.DeWitt. Buffer-Pool Aware Query Optimization. In Proceedings of CIDR 2005.

[15] F.Reiss, T. Kanungo. A Characterization of the Sensitivity of Query Optimization to Storage Access Parameters. In Proceedings of ACM SIGMOD 2003.

[16] U.Srivastava et al. ISOMER Consistent Histogram Construction using Query Feedback. In Proceedings of ICDE 2006.

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

[18] B.T. Vander-Zanden, H.M.Taylor, D.Bitton. Estimating Block Accesses When Attributes are Correlated. In Proceedings of VLDB 1986.

[19] J.S. Vitter. Random Sampling with a Reservoir. ACM Transactions on Math. Software. 11(1): 37-57 (1985)

[20] K. Whang, B.T. Vander-Zanden, H.M. Taylor. Linear Time Probabilistic Counting Algorithms for Database Applications. ACM Transactions on Database Systems (TODS) Volume 15 , Issue 2 (June 1990).

[21] Microsoft SQL Server 2005 Product Documentation. http://msdn.microsoft.com.