Cardinality Estimation Using Sample Views with Quality Assurance

Per-Ake Larson
Microsoft Research
palarson@microsoft.com

Wolfgang Lehner *
Dresden Univ. of Tech.
wolfgang.lehner@tu-dresden.de

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

Peter Zabback
Microsoft
pzabback@microsoft.com

ABSTRACT

Accurate cardinality estimation is critically important to high-quality query optimization. It is well known that conventional cardinality estimation based on histograms or similar statistics may produce extremely poor estimates in a variety of situations, for example, queries with complex predicates, correlation among columns, or predicates containing user-defined functions. In this paper, we propose a new, general cardinality estimation technique that combines random sampling and materialized view technology to produce accurate estimates even in these situations. As a major innovation, we exploit feedback information from query execution and process control techniques to assure that estimates remain statistically valid when the underlying data changes. Experimental results based on a prototype implementation in Microsoft SQL Server demonstrate the practicality of the approach and illustrate the dramatic effects improved cardinality estimates may have.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*Query Processing*

General Terms

Algorithms, performance

Keywords

Query optimization, cardinality estimation, sample views, sequential sampling, statistical quality control

1. INTRODUCTION

To reliably produce efficient execution plans a query optimizer needs accurate cardinality estimates. Cardinality estimation in commercial database systems relies on statistics, primarily single-column histograms, computed from base data or materialized views. As is well-known, this approach may produce estimates that are orders of magnitude off. Traditional estimation methods perform poorly in several situations.

Jagged distributions: A histogram has a limited resolution to approximate an underlying distribution. The actual distribution of data may be much more jagged than what can be represented by a histogram.

Complex predicates: Queries with complex predicates containing combinations of `AND`, `OR`, `NOT`, and `IN`.

Functions: For predicates or grouping expressions containing built-in or user-defined functions commercial systems frequently give up and apply some "magic numbers" to come up with an estimate.

Correlation: Correlation among columns, in the same table or in joined tables, can have a dramatic effect on cardinalities. Traditional estimation methods assume independence between columns but, when this does not hold, estimates may be way off. Consider, for example, the predicate (COLOR ='white' and PRODUCT = 'iPod').

Error propagation: Cardinality estimation proceeds bottom up in an expression. After combining column-wise estimates to compute the number of rows coming from a single table, the local cardinalities are combined and propagated up to the root operator. For large queries, the errors produced by this propagation process can be extremely large.

Stale statistics: Poor estimates are often caused by stale statistics. If the underlying data changes, a histogram may need to be refreshed to accurately reflect the data. Current policies for triggering refresh of statistics are usually heuristic and not based on sound statistical criteria and thus may result in stale statistics or unnecessarily frequent updates. Another reason for incorrect statistics may be that they were computed from a poor (non-random) sample of the data.

In this paper, we propose a novel method for cardinality estimation based on sample views. A sample view is a materialized view that contains just a random sample of the result. A sample view is not continuously maintained when the underlying tables are updated – it's too expensive – but refreshed when the estimates derived from it are no longer statistically valid. We apply statistical process control techniques to avoid unnecessary refresh activity while still ensuring the quality of estimates.

As illustrated in Figure 1, our approach to cardinality estimation using sample views consists of two conceptual loops: an exploitation loop and a feedback loop.

Sample views are exploited during query optimization to compute cardinality estimates. Suppose we have created a sample view SV over the join expression $SV = R \bowtie_{p_1} S \bowtie_{p_2} T$. During optimization of a query we may need to estimate

*Work performed while a visiting researcher at Microsoft.

SIGMOD'07, June 11–14, 2007, Beijing, China.

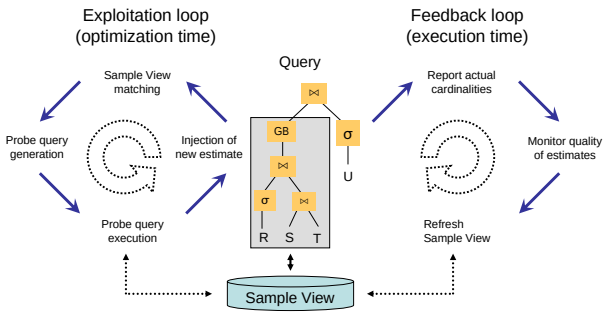


Figure 1: Conceptual overview of our approach

the cardinality of an expression $E = \sigma_{p_r}(R) \bowtie_{p_1} S \bowtie_{p_2} T$. Normal view matching determines that E can be expressed in terms of the view as $E = \sigma_{p_r} SV$. The cardinality of E is then estimated by executing the probe query $\sigma_{p_r} SV$ against the sample and counting the number of rows returned. The probe query scans the sample sequentially and stops as soon as the estimate has reached the desired accuracy. Finally, the estimate is injected into the optimizers data structures and optimization continues.

The feedback loop monitors the quality of estimates while queries are executing and triggers refresh of a sample when its estimates are no longer statistically valid. Suppose query Q contains expression E above and was optimized using estimates computed from sample view SV. Every time query Q executes, it reports the number of actual rows for expression E to a quality control component where the estimate is compared to the actual value and the estimation error is normalized. Multiple queries may produce feedback on estimates based on SV, producing a stream of normalized errors. We apply statistically sound process control techniques to monitor the stream of normalized errors associated with SV and signal the need to refresh the sample.

Sample views are intended to augment, not replace, conventional cardinality estimation. Even though probe queries run fast, they do increase optimization time. The overhead doesn't much matter for expensive queries that run for several seconds or even minutes but it may be too high for very cheap queries.

This work is based on the premise that better cardinality estimates result in better query execution plans. While this holds in general, it is not guaranteed for every single query expression. However, experience strongly suggests that commercial-grade optimizers seldom produce a truly poor plan when accurate cardinality estimates are available.

The main contributions of our work are as follows.

1. We combine random sampling and materialized view technology to provide cardinality estimates with statistically guaranteed accuracy.

2. We provide probe query patterns dependent on the type of cardinality estimate requirement (number of qualifying tuples, number of distinct values and a combination).

3. We apply sequential sampling to reduce the overhead of probe queries while still satisfying stated accuracy requirements.

4. We exploit query feedback and statistical process control techniques to trigger refresh of a sample only when it no longer produces statistically correct estimates.

5. We demonstrate the practicality of our approach by experiments using a prototype implementation in Microsoft SQL Server.

The rest of the paper is organized as follows. We begin by describing how sample views are defined, created and stored in Section 2. In Section 3 we describe how sample views are used for cardinality estimation during optimization and, by means of an example, show the dramatic impact that improved estimates may have. In Section 4 we describe how sample views are maintained and our use of statistical process control. Related work is summarized in Section 5. We conclude in Section 6.

2. SAMPLE VIEW REPRESENTATION

In this section we briefly outline how sample views are defined, how they are stored, and how the sample is computed.

Regular materialized views are used during execution of a query to speed up processing of (usually) expensive queries. At optimization time, the optimizer performs view matching, constructs a substitute expression and decides whether or not to use a materialized view. Sample views are used *only* at optimization time. After successful matching of a sample view, the optimizer may run a probe query against the view to compute a cardinality estimate.

View definition

A sample view is defined in the same way as a regular materialized view and the same restrictions apply. The view definition may contain any predicates except subqueries in the where clause and/or grouping expressions in the group-by clause. The general form of a view definition is as follows.

```
CREATE VIEW <svname> WITH SCHEMABINDING AS
SELECT <keycols>, ...
FROM ...
[WHERE ...]
[GROUP BY ...]
```

We envision that most sample views will contain only joins and no further restrictions because such views are more versatile than views with restrictions. Similarly, the more columns the view outputs, the more broadly useful it is.

Like a regular materialized view, a sample view is materialized only when we create a clustered index on the view.

```
CREATE UNIQUE CLUSTERED INDEX <clidxname> ON <svname>
      (<keycols>) [ROWSAMPLE <samplepercent> PERCENT]
```

The rowsample clause marks the view as a sample view and prevents it from being used as a regular materialized view. The parameter <*samplepercent*> specifies what fraction of the view to include in the sample.

The following example creates a 2% sample view over the join of lineitem and orders with a receipt date during 1993.

```
CREATE VIEW SV_LINEITEM_ORDERS WITH SCHEMABINDING AS
SELECT L_ORDERKEY, L_PARTKEY, L_SUPPKEY, L_LINENUMBER, ...
       O_ORDERSTATUS, O_TOTALPRICE, O_ORDERDATE, ...
FROM TPCD.LINEITEM, TPCD.ORDERS
WHERE L_RECEIPTDATE BETWEEN '1993-01-01' and '1993-12-31'
  AND L_ORDERKEY = O_ORDERKEY
```

```
CREATE UNIQUE
CLUSTERED INDEX SV_LINEITEM_ORDERS_CLIDX ON
   SV_LINEITEM_ORDERS (L_ORDERKEY, L_LINENUMBER)
   ROWSAMPLE 2 PERCENT
```

Internal representation

The internal representation of a sample view contains an additional column __RAND of type short int. This column holds a random value drawn from a uniform distribution in the range $[0; MAXRAND]$. For reasons that will become clear later, $MAXRAND$ should not be too large, say, at most 1000.

Sample views automatically add the __RAND column as the first key column to physically store the rows sorted on the random value. That is, internally the clustering index is defined as

```
CREATE CLUSTERED INDEX <clidxname>
     ON <svname> (__RAND, <keycols>)
```

This trick makes it possible to apply sequential sampling, as described in more detail in the next section. Rows containing the same __RAND value are clustered together. If a sequential scan of the sample is terminated at the end of a cluster, the set of rows scanned is a statistically valid, simple random sample. However, this cannot be guaranteed if the scan is terminated within a cluster. Within a cluster the rows are sorted on <*keycols*> and the ordering may be correlated with selectivity. For example, rows early in the cluster may be more likely to satisfy the query predicate than rows later in the cluster.

Sampling scheme

To be able to apply a large class of estimators, sample views contain simple random samples created by Bernoulli sampling. When the view expression is evaluated, each output row is randomly selected for the sample with the probability specified in the ROWSAMPLE clause. Hence, the actual percentage of rows in the sample may be slightly different than specified. The total number of rows in the view and number of rows in the sample are recorded in the catalog. In subsequent formulas we denote them by N_v and N_s.

3. EXPLOITING SAMPLE VIEWS

Sample views can be used to compute cardinality estimates for SPJG-expressions, that is, select-project-join expressions with at most one group-by on top. Different estimators are used for expressions with and without aggregation. This section describes the estimators used and how to construct probe queries computing the estimates.

The matching process for sample views is the same as for regular materialized views. However, for a sample view a regular substitute expression is not generated but instead expression fragments that are used later to construct the appropriate probe query. The overhead of matching sample views is the same as for regular materialized views and usually negligible.

3.1 Sequential sampling

Executing a probe query during optimization increases the optimization time. The overhead can be reduced greatly by using *sequential sampling* [28] (also known as *progressive sampling*). The basic idea of sequential sampling is to process only as many rows of the sample as is needed to compute a sufficiently accurate estimate. In many cases, only a small subset of the rows is needed to reach the desired accuracy.

To facilitate sequential sampling we store the sample in sorted order on the __RAND column. Rows with the same __RAND value are thus clustered together. If we scan the sample sequentially and terminate the scan at the end of a cluster, the set of rows scanned is a statistically valid, simple random sample.

Unfortunately, we cannot decide when constructing a probe query how many sample rows are required to achieve the desired accuracy. Instead, we construct probe queries that evaluate a stopping condition at every break in the __RAND column and terminates the scan as soon as the condition is satisfied. The stopping condition primarily tests whether the (estimated) standard error of the estimate is within the specified range. It must also correctly handle two "end cases": ensure that an estimate is returned when the whole sample has been scanned (even if the desired accuracy has not been reached); and ensure that the estimate is based on a minimal set of evidence, that is, not allowing the scan to stop too early. The exact form of the condition depends on the estimator used and will be described later.

We do not want to evaluate the stopping condition very frequently because it is somewhat complex and expensive to compute. We handle this by generating values for the __RAND column from a smallish range and constructing probe queries that test the stopping condition only once for each cluster of rows with the same __RAND value. For example, generating values in the range [1,100] means that the stopping condition is evaluated at most 100 times. We also have to make sure that execution plans for probe queries do not contain any blocking operators (such as a sort), that is, operators that consume the whole input before producing output tuples.

3.2 Selectivity estimation

Consider a query expression $E = \sigma_{p_q}(T_1 \times \cdots T_n)$ and a (sample) view defined by $V = \sigma_{p_v}(T_1 \times \cdots T_n)$. The view matches the query expression if and only if $p_q \Rightarrow p_v$, that is, every row that satisfies the query predicate also satisfies the view predicate. If this is the case, the view matching procedure returns a residual predicate p_r with the property that $p_q = p_r \wedge p_v$. If the view were fully materialized, the query expression could then be computed from the view by the substitute expression $E = \sigma_{p_r} V$. We use the sample to estimate the cardinality of the expression $\sigma_{p_r} V$, which of course also is an estimate of the cardinality of the original query expression $\sigma_{p_q}(T_1 \times \cdots T_n)$.

Estimator

Provided the view outputs all columns referenced in p_r, we can run the probe query $\sigma_{p_r} V$ against the whole sample or any random subset thereof. Suppose we run the probe query on a subset containing n_s ($n_s \leq N_s$) rows and n_p rows satisfy the residual predicate p_r. We then estimate the selectivity of predicate p_r as $\hat{P} = n_p/n_s$ with an (estimated) standard error of $\hat{s} = \sqrt{\hat{P}(1-\hat{P})/n_s}$. Finally, the cardinality of E can then be estimated as $Card(E) = \hat{P}N_v$ with an (estimated) standard error of $\hat{S} = \hat{s}N_v$. We assume that the sample is a sufficiently small that the finite population correction (fpc) can be ignored.

Probe queries

If we are willing to scan the complete sample, the probe query is straightforward.

```
SELECT COUNT(*) AS n_p FROM <view name> WHERE <pred>
```

<*pred*> is the placeholder for the residual predicate, <*view name*> for the name of the sample view. The sample size N_s is available in the catalog so the estimate can be compute as $\hat{P} = n\_p/N_s$.

For sequential sampling, a considerably more complex probe query is needed because the query must return both the number of rows scanned and the number of rows satisfying the predicate, use a sequential scan, and terminate as soon as the desired accuracy has been reached.

```
SELECT TOP(1) SUM(t2.cnt_p) AS n_p,
              SUM(t2.cnt_all) AS n_s
FROM (SELECT __RAND,
        SUM(t1.cnt_p) AS cnt_p, COUNT(*) AS cnt_all
      FROM (SELECT __RAND,
              CASE WHEN <pred> THEN 1 ELSE 0 END AS cnt_p
            FROM <view name> ) t1
      GROUP BY __RAND) t2
GROUP BY ALL WITH STEPWISE
HAVING <stopping condition>
OPTION (ORDER GROUP)
```

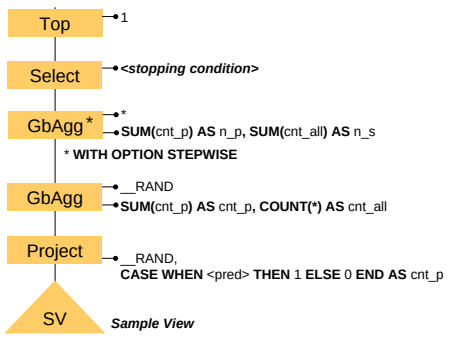


Figure 2: Execution plan of probe query for selectivity estimation

The execution plan generated for this query is shown in Figure 2. We briefly describe the function of each operator in the plan.

The first (lowest) operator evaluates the residual predicate and sets column *cnt_p* to one if it is satisfied and otherwise to zero.

The second operator performs a preaggregation by counting the overall number and the number of qualified rows for the same __RAND value. Since the sample view is sorted on the __RAND column, aggregation can be done by a streaming aggregate operator that outputs a row as soon as it encounters a break in __RAND values. The purpose of the preaggregation is to reduce the number of times the stopping condition is evaluated.

The third operator performs the final aggregation by summing up the partial results cumulatively. The group-by operator runs with option STEPWISE[1] enabled, which causes the operator to generate an output row with the current state of the aggregate values for every incoming row. Since the scalar aggregation in this particular setup does not specify any grouping columns, we use the keyword ALL as a placeholder.

The final select operator applies the stopping condition as specified in the HAVING clause of the probe query.

The last operator is a TOP operator that closes the scan after having output one row. This row is the first row produced that satisfies the stopping condition, that is, it contains an estimate that satisfies our accuracy requirements.

The stopping condition requires that a certain minimum number of rows be read to have enough evidence. Once this number has been reached, scanning stops as soon as one of the following three criteria is satisfied.

1. *Relative Error:* the 95% confidence interval is less than 10% of the point estimate.

2. *Absolute Error:* the estimate number of qualifying rows (in the real data set) is 10 or less with 90% confidence.

3. *End of sample:* the end of the sample is reached.

This probe query and execution plan meet the goals stated above. The stopping condition is evaluated only after combining sample rows with the same __RAND value. The stepwise option causes the grouping operator to output its running results. The final TOP operator shuts down the query as soon as an estimate with acceptable accuracy is available. Finally, keeping the rows sorted in "random order" ensures that the estimate produced when the scan stops is based on a simple random sample.

3.3 Estimating the Number of Distinct Values

We now consider query expressions with aggregation. Consider a query expression $E = \gamma^A_G \sigma_{p_q}(T_1 \times \cdots T_n)$ and a (sample) view defined by $V = \sigma_{p_v}(T_1 \times \cdots T_n)$. The notation γ^A_G specifies grouping with grouping expressions G and aggregation functions A. If the view matches, both further selection and further aggregation may be needed. View matching returns, in essence, the operators $\gamma^A_G \sigma_{p_r}$. If the view were fully materialized, the query expression could be computed from the view by the expression $E = \gamma^A_G \sigma_{p_r} V$. We use the sample to estimate the cardinality of the expression $\gamma^A_G \sigma_{p_r} V$, which of course also is an estimate of the cardinality of the original query expression.

Estimators

Estimating the cardinality of $\gamma^A_G \sigma_{p_r} V$ is the same as estimating the number of distinct values occurring in the result of $\pi_G \sigma_{p_r} V$. This is an old statistical estimation problem, originally formulated as estimating the number of species in a population, and many different estimators have been proposed. We have experimentally evaluated many of them and three of them stood out as being most robust: Chao's estimator [6], Shlosser's estimator [27], and the Poisson estimator [30, 25]. All other estimators that we evaluated were much more erratic, sometimes producing good estimates but sometimes being wildly off the mark.

We did not want to commit to a specific estimator or combination of estimators at this stage. Instead, we use a probe query that returns data needed by a variety of operators, giving us the freedom to decide what estimator to

[1]The STEPWISE option is only available for system-generated queries and not for regular queries.

use later. Virtually all distinct-value estimators take as input value pairs consisting of (group size, number of groups of that size) computed from the sample. We designed the probe query to return this information in a single tuple because we want to apply sequential sampling and terminate as soon as a sufficiently accurate estimate is returned. (The estimator we use for testing purposes is a combination of the three estimators mentioned above.)

Probe query

The following probe query outputs the data needed by many distinct-value estimators. <*pred*> is the residual predicate, <*grp-list*> is a placeholder for the list of grouping columns returned by view matching. PICKPIVOT is a fixed small table; its contents and use will be explained shortly. The execution plan generated for this query is shown in Figure 3.

```
SELECT TOP(1)
       SUM(t2.grpcard) AS dvcard,
       SUM(t2.grpcard * t2.grpsize) as scard
       SUM(t2.grpcard * pt.flag1) AS grp1card,
       ...,
       SUM(t2.grpcard * pt.flagn) AS grpncard
FROM (SELECT smplsize, t1.grpsize, COUNT(*) AS grpcard
      FROM (SELECT INPUTCOUNT() as smplsize, <grp-list>,
                   COUNT(*) AS grpsize,
            FROM <svname> WHERE <pred>
            GROUP BY <grp-list> WITH OPTION FLUSH(__RAND)
            OPTION (HASH GROUP)) t1
      GROUP BY smplsize, t1.grpsize
      OPTION (ORDER GROUP)) t2,
      PICKPIVOT pt
WHERE t2.grpsize = pt.id
GROUP BY smplsize
HAVING <stopping condition>
```

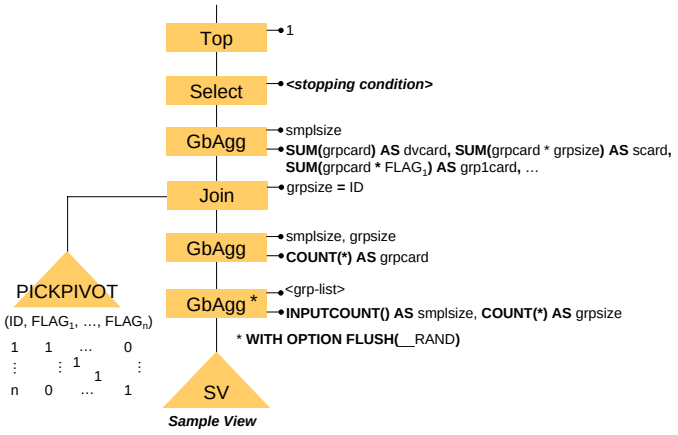


Figure 3: Execution plan of probe query for estimating the number of distinct values

The first (lowest) operator computes the group size for different group expression values within each partition defined by the _RAND value. Similar to the option STEPWISE for sort-based aggregation, we need to add an option FLUSH(<column list>) to the hash-based aggregation operator. When option FLUSH is enabled, the operator performs its normal aggregation and, in addition, outputs the current state of aggregation whenever the value of the flush indicator columns <column list> changes (but does not otherwise change the state). The function INPUTCOUNT() returns the number of input rows consumed so far by the operator. We include this in every output row so we can tell which tuples originate from the same subsample. The second aggregation simply counts the number of groups of each size.

We now have the result we need but it consists of multiple rows and we want the result as a single row, that is, we need to pivot the result. The next two operators perform the pivoting using the Rozenshtein method. The join operator joins the result with a static table PICKPIVOT that consist of n rows with an ID column and n additional columns, labelled flag1, flag2, and so on. Row $i, i = 1, 2, \cdots n$ has the value i in the ID column, one in column flag$i$, and zero in all other columns. The join extends each row with n flag columns and column flag$i$ contains a one if the row represents groups of size i. The group-by operator completes the pivoting by constructing a single output row for each sample size.

The pivot step collected all the group cardinalities together, producing a single row for each sample size. The stopping condition is then evaluated to determine whether an estimate meeting the accuracy can be computed. Finally, the TOP(1) operator closes the query after outputting the first row satisfying the stopping condition.

In the same way as for selectivity estimation, stopping conditions for distinct-value estimation should be tied to the estimate's confidence interval. Unfortunately, the calculations required to compute confidence intervals for general distinct-value estimators like Shlosser's estimator [27] or jackknife-style estimators [16] are quite complex and beyond what can be done in SQL. We currently use a stopping condition that combines two factors.

1. *Confidence interval of Poisson estimator:* As long as the sample is a small fraction of the population, the Poisson estimator generally provides a stable estimate and its confidence intervals are relatively easy to compute [30, 25]. Hence, our first stopping condition is that the 95% confidence interval of the Poisson estimator is less than 10% of its point estimate.

2. *Distance between Chao and Shlosser estimator:* The Chao estimator [6] is a lower-bound estimator while the Shlosser estimator [27] tends to overestimate the actual value. For larger sampling fractions, scanning stops when the (relative or absolute) difference between these two estimates becomes less than a target value.

Figure 4 illustrates the execution of the above probe query. When the _RAND column changes from 4711 to 4712, the first group-by operator outputs two rows, (3,'a',1) and (3,'b',2). There are two groups, one of size one and one of size two, so the second group by operators outputs two rows, (3,1,1) and (3,2,1), which the pivoting combines into a single row. Scanning continues with the next cluster of rows, with _RAND value 4712, and added to the existing state of the first group-by operator. When the end of the cluster is reached, there are two groups of size 2 and two groups of size one. The operator outputs two rows, (6,1,2) and (6,2,2), which again are combined into a single row by the pivoting. This process continues until the stopping condition is satisfied.

3.4 Experiments

The following query against a 1GB TPC-H database demonstrates the benefits of sample views and sequential sampling.

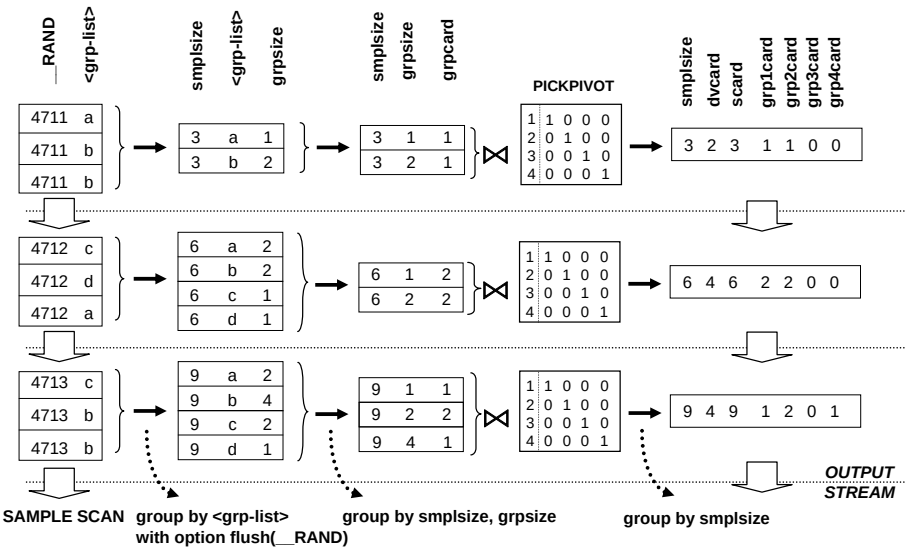
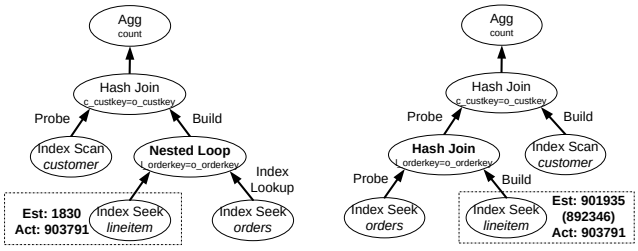


Figure 4: Example execution of probe query for distinct-value estimation

```
SELECT c_nationkey, count(*)
FROM lineitem, orders, customer
WHERE l_receiptdate < DATEADD(day, 30, l_shipdate)
  AND l_commitdate < DATEADD(day, 30, l_shipdate)
  AND l_commitdate < DATEADD(day, 30, l_receiptdate)
  AND l_receiptdate > '1996-01-01'
  AND l_commitdate > '1996-01-01'
  AND l_shipdate > '1996-01-01'
  AND l_quantity > 25
  AND l_orderkey = o_orderkey
  AND c_custkey = o_custkey
GROUP BY c_nationkey
```

The query contains a complex predicate on the `lineitem` table. Columns l_receiptdate and l_commitdate are heavily correlated but the regular cardinality estimation assumes that they are independent. It estimates that only 1,830 out of approximately 6 million `lineitem` rows satisfy the predicate, which underestimates the actual number of 903,791 rows by more than two orders of magnitude.



(a) Original Plan (b) Using Sample Views

Figure 5: Query Plans

The query plan generated by the optimizer based on the incorrect estimate is shown in Figure 5 (a). The plan makes sense under the assumption that only a small number of `lineitem` rows qualify. The few rows are joined to `orders` by a nested-loop join. The result is still small, compared with `customer`, and thus used to build a hash table while the larger `customer` table probes the hash table. The join result is aggregated to obtain the final result. However, the actual number of qualifying `lineitem` rows is large, making the nested-loop join a very poor choice.

To improve cardinality estimation we created a sample view `sv_lineitem` containing a 1% sample of `lineitem` (59,538 rows). This sample view was used to estimate the number of qualifying `lineitem` rows, using both a full scan of the sample and sequential sampling. Due to space limitation, we do not list the probe queries. Table 1 summarizes the cardinality estimates obtained and the number of sample rows scanned.

	Original	Using sample view	
		Full sample	Sequential
Estimate	1830	901,935	892,346
Sampled rows	N/A	59,538	5,974

Table 1: Estimated number of `lineitem` rows satisfying the predicate (actual number is 903,791)

The estimates computed from the sample view are highly accurate, no matter whether we do a full scan or use sequential sampling. Note that the sequential sampling stops after scanning only 5,974 rows, which is about 1%% of the rows in the `lineitem` table.

With the more accurate cardinality estimates available, the optimizer produces a different plan, which is shown in Figure 5 (b). A hash join between `lineitem` and `orders` is used instead of a nested-loop join. Since the result is larger than the `customer` table, the `customer` table is used as the build input.

So far we have shown that using the sample view dramatically improves the accuracy of the estimate, which affects the choice of query plan. Table 2 summarizes the effects on optimization and execution times.

The revised plan achieves a 4-fold speedup in execution time. A full scan of the sample view increased the optimization time by about 50% but the overhead is negligible compared with savings in execution time. Sequential sam-

	Original	Using sample view	
		Full	Sequential
Optimization (secs)	0.095	0.153	0.120
Execution (secs)	14.462	3.891	

Table 2: Optimization and execution times

pling cut the optimization overhead in half without losing any benefit. The cost of sequential sampling does not necessarily grow with the sample view size because scanning stops as soon as the desired accuracy has been reached.

4. MAINTAINING SAMPLE VIEWS

To produce accurate estimates a sample view must contain a statistically valid random sample of the view result. As the data in the underlying base tables changes, we must ensure that this property still holds. The simplest approach is to treat sample views in the same way as regular materialized views and maintain them incrementally. However, we rejected this approach for two reasons.

1. Experiments showed that the overhead is simply too high, especially if the view consists of multiple joins. Even after applying a variety of tricks, we still saw overhead of 5-10% for updating a single view joining three tables.

2. Incremental view maintenance is performed as part of the update transaction. The presence of sample views would then impact overall system load, even during peak times. We want to exploit "free cycles" during low system load for maintenance activity.

Instead we propose a full refresh strategy, that is, the sample is not maintained but instead completely recomputed when required. This immediately raises the question of when to trigger a refresh. Our core idea is to tie refresh decisions to the statistical validity of the sample. We use feedback from running queries to check whether estimates computed from the sample are still within statistical error bounds and defer refresh as long as the sample *statistically* still represents the underlying data. Refresh decisions are not based on simple heuristics but rely on well-proven process control techniques based on sound statistical principles.

4.1 Quality Control Process

Our quality control process is illustrated in Figure 6 in more detail.

During optimization of a query, the cardinality of some subexpressions may be estimated from matching sample views. Some, not necessarily all, of those subexpressions may be present in the final execution plan. Our goal is to have every such subexpression report back to its sample view the actual number of rows so the accuracy of the estimate can be assessed. A sample view V may have many different queries returning feedback, some with low selectivity and some with high selectivity. Estimation errors from different queries cannot be compared directly so we first convert them to a common scale. In this way, each sample view receives a stream of normalized estimation errors during query execution. We apply standard statistical process control techniques to the stream of normalized errors to detect when the errors are no longer within statistical bounds, thereby triggering refresh of the sample in the view.

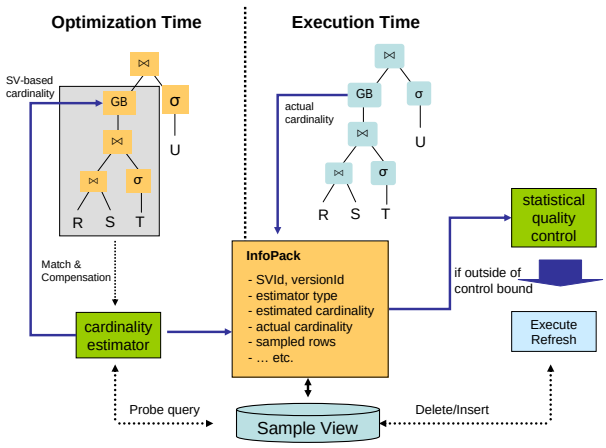


Figure 6: Quality control process

Feedback collection

When a cardinality estimate for a subexpression E_i is computed from a sample view during optimization, we attach an InfoPack structure to the subexpression. More precisely, it is attached to E_i and all expressions equivalent to E_i. If E_i or any expression equivalent to E_i appears in the final plan, we attach the InfoPack to the operator in the execution plan that corresponds to the root of E_i. During execution the operator counts the number of tuples it outputs and, provided that it has a valid count at the end, uses the information in the InfoPack to report back to the appropriate sample view.

The InfoPack structure contains the following fields.

1. the ID of the sample view used to compute the cardinality estimate

2. the version number of the sample used (explained in Section 4.2)

3. a flag indicating the type of estimator used, f

4. the estimated number of rows produced by the operator, $\hat{K}$

5. the actual number of rows produced by the operator, K, (filled in after execution)

6. the number of rows from the sample read by the probe query, n

If the operator has a valid row count when execution finishes, the actual cardinality is filled in and the InfoPack is handed over to the quality control component. A row count is not valid unless the operator has seen the end of input, which it may not do if, for example, the query is terminated early by a user or by a top operator.

Normalizing estimation errors

Let $\mathcal{S}$ denote a random sample of size n drawn from the result of view V (population) with N_v rows ($N_v \gg n$). The cardinality estimate $\hat{K}$ is a function of $\mathcal{S}$, that is, $\hat{K} = f_e(\mathcal{S})$ where f_e is the estimator function. $\hat{K}$ is a stochastic variable with a well-defined probability density function (PDF) that depends on the estimator f_e and the sampling scheme. Our sampling scheme is the simplest possible: random samples

of size n are drawn from a population of size N_v. Denote the probability density function of $\hat{K}$ for this scenario by $P(\hat{K} = x) = g_{f_e}(x, n, N_v)$ and the cumulative density function (CDF) by $P(\hat{K} \leq x) = G_{f_e}(x, n, N_v)$.

As queries execute, we receive a stream $R_1, R_2, \cdots$ of actual cardinalities corresponding to estimates computed from view V. R_i is not a simple value but a tuple consisting of $(K_i, \hat{K}_i, f_i, n_i)$ where K_i is the actual cardinality, $\hat{K}_i$ is the estimate, f_i is a flag indicating which estimator was used, and n_i is the sample size used when computing the estimate.

The estimation errors are easy to compute, $\hat{K}_i - K_i$ but, unfortunately, they cannot be compared directly against each other because the estimates may be based on different estimators and different sample sizes.

We convert the errors to a common scale by mapping $\hat{K}_i$ through the estimator's CDF, that is, we compute $\hat{y}_i = G_{f_i}(\hat{K}_i, n_i, N_v)$. Because the mapping is through the estimator's CDF, $\hat{y}_i$ is, by definition, uniformly distributed in $[0, 1]$ with expected value $E(\hat{y}_i) = 0.5$ and variance $S^2(\hat{y}_i) = 1/12$. If the estimator is unbiased, then $G_{f_i}(K_i, n_i, N_v) = 0.5$. The normalized error is computed as $e_i = G_{f_i}(\hat{K}_i, n_i, N_v) - G_{f_i}(K_i, n_i, N_v) = \hat{y}_i - 0.5$. e_i is uniformly distributed in $[-0.5, 0.5]$. This error normalization process calibrates an error relative to the accuracy of the estimator used.

To actually implement the normalization process we need the cdfs for the estimators used. For selectivity (proportion) estimation, the density function of the estimate is a binomial distribution with parameters $p = K/N_v$ and n, that is,

$$Bin(x, K/N_v, n) = \binom{n}{x} (K/N_v)^x (1 - (K/N_v))^{n-x}.$$

The function returns the probability that exactly x rows in a random sample of n rows satisfy the residual predicate when the actual fraction of all rows (the population) satisfying the predicate is K/N_v. For mapping purposes we need the cumulative binomial distribution

$$CumBin(x, K/N_v, n) = \sum_{i=0}^{x} Bin(i, K/N_v, n).$$

This formula is inefficient and not useful for actual computation. The cumulative binomial distribution can be expressed in terms of the Incomplete Beta function (see [1], formula 26.5.24 on page 945), which is more efficient computationally. The binomial distribution $Bin(x, p, n)$ can also be approximated by a normal distribution $Norm(x, np, np(1-p))$ provided that np and $n(1-p)$ are not too small ($np \geq 10$ and $n(1-p) > 10$).

Figure 7 illustrates the normalization process for two different predicates, one with selectivity 10% and one with 55%. An exact estimate ($\hat{K}_i = K_i$) would result in $e_i = (0.5 - 0.5) = 0$. Overestimating the selectivity as 61% when the actual selectivity is 55%, yields a normalized error of $e_i = (0.89 - 0.5) = 0.39$. Underestimating it as 9% when the actual rate is 10% produces a normalized error of $e_i = (0.38 - 0.5) = -0.12$.

For distinct-value estimators, the exact distribution is often not known but the expected value and variance usually are. For many of the estimators, the distribution can be approximated by a normal distribution.

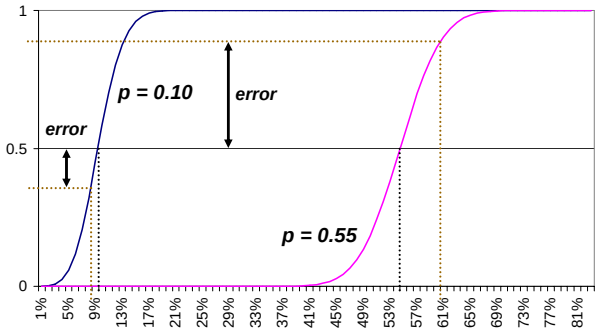


Figure 7: Normalization of observed estimation errors

Process control

Each sample view V receives a stream of normalized estimation errors. As long as its sample remains a statistically valid random sample of the underlying view, the normalized errors are uniformly distributed in $[-0.5, 0.5]$. We apply standard statistical process control techniques, see [22], to monitor whether the quality of the estimates computed from the sample are still under control, that is, still within statistically acceptable bounds.

To apply process control techniques one must decide what variable to monitor and determine its probability distribution. Instead of monitoring the normalized errors e_i directly, we monitor the variable $z_i = 2|e_i|$. We take the absolute value because we do not want negative and positive errors to cancel each outer out. The variable $|e_i|$ is uniformly distributed in $[0, 0.5]$ and multiplying it by two gives us a variable that is uniformly distributed in $[0, 1]$.

To smooth out random variations, we compute an exponentially weighted smoothed average (EWMA) of the monitored variable z_i. When observation z_i arrives, the average is updated as follows

$$E_{EWMA} := \alpha * z_i + (1 - \alpha) * E_{EWMA}.$$

The constant α is typically small, in the range of 0.05 or less. The higher the value, the more sensitive the average is to changes in the underlying data.

The purpose of process control is to trigger corrective action when the exponentially smoothed average drifts outside of an *a priori* defined control interval. The interval can be one-sided or two-sided depending on the process. The control bounds are set so that the probability of remaining within the control interval is high, in the order 0.99999 to 0.999999, as long as the process is statistically stable. We use a one-sided interval because we only need to guard against errors being too large.

It can be shown [22] that E_{EWMA} has a normal distribution with, in our case, expected value 0.5 and standard deviation $\sigma_{EWMA} := 2 * \sqrt{(var_{uniform} * f_{EWMA})} = 2 * \sqrt{(1/12 * \alpha/(2 - \alpha))}$. The factor $f_{EWMA} = \alpha/(2 - \alpha)$ shows the dampening effect of the exponentially smoothing. For example, with $\alpha = 0.04$, we have $\sigma = 0.082$. If we want the probability of remaining within the control interval to be 0.99999, we set the control bound to 0.5 + 4.265*0.082 = 0.85.

Example

Figures 8 and 9 illustrate our quality control mechanism in action for the following simple scenario. The same query runs repeatedly. The selectivity of the query predicate was estimated from a sample view so the query reports the actual cardinality each time it runs. At iteration 16, an update transaction begins modifying the underlying base data in a way that has significant impact on the selectivity of the query. To demonstrate the effect more clearly, the queries run in read-uncommitted mode. After the start of the update, the queries begin reporting larger and larger errors, very quickly reaching the maximal value of 1.0. The exponentially smoothed average increases more slowly but as soon as it crosses the control bound, a refresh of the sample view is scheduled and (in this particular case) immediately executed. Queries continue to run while the sample is being refreshed and the average keeps on increasing. When the refresh is finished, E_{EWMA} is reset to its expected value of 0.5 and monitoring begins again. The query is also reoptimized and obtains a new estimate. Subsequent executions (after step 31) actually report a more accurate estimate than before the refresh.

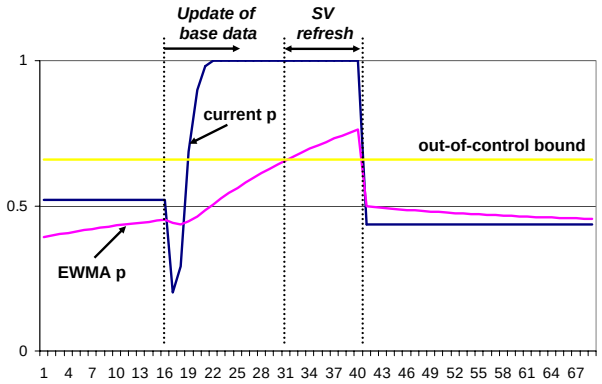


Figure 8: Reaction of control mechanism to a sudden change in data

Figure 9 shows how the actual selectivity of the query predicate slowly changes are base data is updated. This change is reflected in the sample only later when the sample has been refreshed.

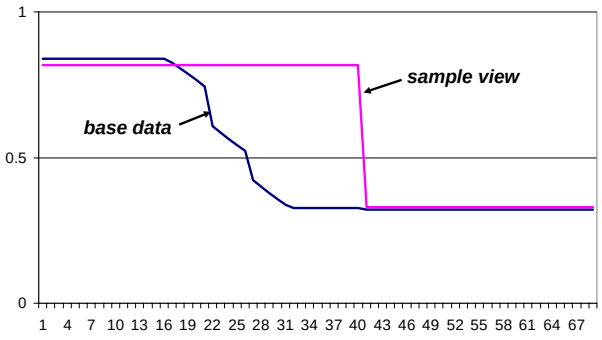


Figure 9: Actual query selectivity and estimate computed from sample view

4.2 Sample refresh procedure

A sample refresh is initiated by putting the sample view into *refresh pending mode* and creating a low-priority background job to carry out the refresh, see Figure 10. While a view is in refresh pending mode, queries may still use it for cardinality estimation (e.g. Q_k in figure 10). However, the cardinality derivation process should be made aware that the estimates may not be accurate.

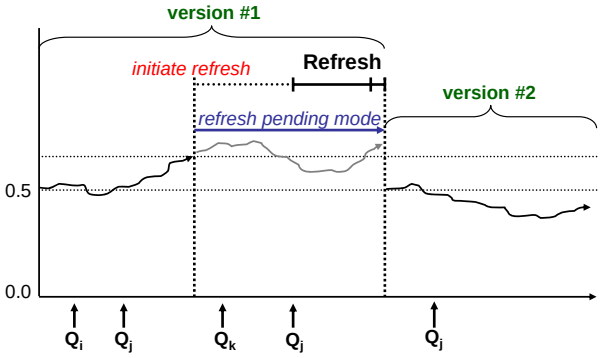


Figure 10: Sample refresh procedure

The background job performing the refresh is divided into two steps to minimize potential lock conflicts. In the first step, the sample is computed in read-uncommitted mode and temporarily stored in a staging table. A second step deletes the old content of the sample view and quickly copies in the sample from the staging table.

When a refresh completes, the version number of the sample is increased and the E_{EWMA} is reset to its expected value 0.5. We do not force recompilation of queries when a sample view is refreshed so queries compiled against an old version of the sample view may still execute and produce feedback information. Feedback from queries with an estimate based on an old version of a sample is ignored. This is the reason for including the sample version number in the InfoPack structure.

4.3 Responsiveness of control system

The statistical quality control mechanism implements a demand-driven refresh schedule. No longer is refresh activity triggered by heuristics based on the number of updates or other simple metrics. Instead, the system adapts automatically to changing data but with a certain delay. How quickly the system responds depends on several factors.

- How rapidly and how much the data changes.
- How much the changes affect the cardinality of running queries. Changes that are not relevant to the query workload don't matter.
- The accuracy of the estimators used. An estimator with a tighter error distribution makes the control system more sensitive to changes.
- The smoothing factor α. The lower the value, the longer it takes for the system to react.
- The control bound. The higher the control bound, the slower the system is to react (but the lower the probability of a false alarm).

Figure 11 illustrates how selectivity changes are reflected in the control variable z_i, that is, in the input to the control system. The example query has a original selectivity of 0.1 against a view of 1 million rows. Because observed estimation errors are normalized using the CDF of the estimator, the sensitivity of the control variable to changes in the selectivity depends on the effective sample size used when computing the estimate. The larger the sample, the tighter the distribution and the higher the sensitivity. Suppose the data is updated and the actual selectivity of the query changes to 0.11. If the estimate was computed from a sample of 1000 rows, this error is translated into a z-value of about 0.7. If the sample size was 10,000, the same error is mapped into a z-value of 1.

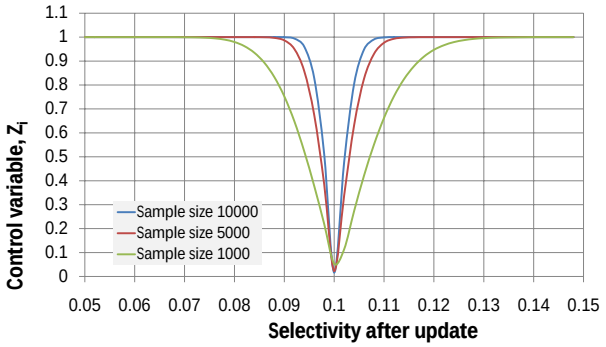


Figure 11: Sensitivity of control variable z_i to selectivity changes

The control system is driven entirely by query feedback so its response time is measured in number of feedback reports received, not in absolute time. This provides an abstract aging model which is determined by the usage and not by the real world time line; a sample view is only as old as the number queries that have exploited it.

Figure 12 shows how the system responds to a sudden change that is large enough to cause all feedback to be mapped into z-values of one. As we saw from the previous example, this does not necessarily require very large selectivity changes; even a change of one or two percentage points may be enough. How quickly the system responds depends on the smoothing constant α and the control bound. In the figure, the control bounds (the dashed lines) are set a 4.265 times the standard error, which corresponds to a risk of a false alarm of 10^{-5}. In this scenario a refresh is triggered after 8 to 13 queries have reported, depending on the value of α.

Figure 13 illustrates the response to a gradual change in data and how it depends on the sample size. In this scenario we have a data set consisting of 1,000,000 rows and a query with a selectivity of 0.1, that is, 100,000 qualifying rows. Between each report the data is modified so that 1,000 additional rows qualify. So after 10 reports, there are 110,000 qualifying rows, after 20 there are 120,000 rows, and so on. The actual selectivity figures are shown across the top of the chart. The figure plots the effect of this gradual change on the exponentially smoothed average ($\alpha = 0.04$) for three different sample size. The larger the samples used, the higher the accuracy of the estimates, and the more rapidly the system reacts to data changes. For a sample size of 5,000, a refresh is triggered when the selectivity has increased by

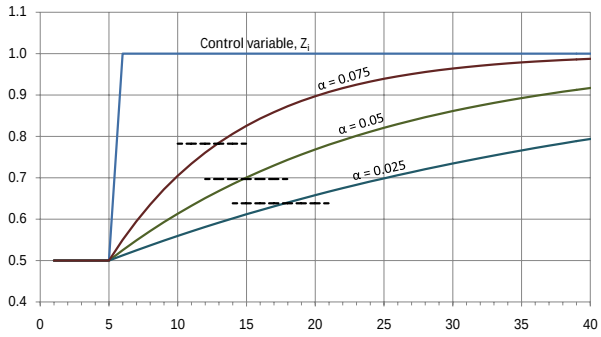


Figure 12: Response of control system to a sudden data change

only two percentage points, from 10% to 12%. This level of sensitivity seems more than adequate; it is unlikely that a cardinality change from 100,000 to 120,000 rows would drastically affect the plan choice.

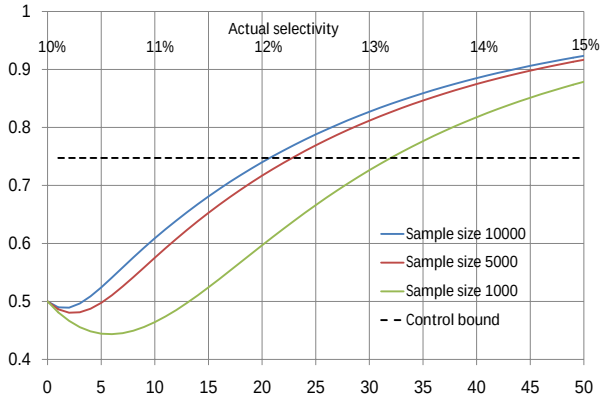


Figure 13: Response of control system to a gradual data change

4.4 Insufficient feedback

A sample view that does not receive sufficient feedback from query execution may become stale without the quality control system noticing. There are several reasons why sufficient feedback may not be forthcoming. If a sample view is not used during query optimization at all, we do not receive any feedback and have no evidence about the statistical accuracy of the sample. However, even if a sample view was used to compute an estimate, there are several reasons why a query may not report any feedback to the view. The subexpression for which the estimate was computed may not be included in the final plan. Even if the subexpression is included in the plan, it may not have a valid count to report at the end because its evaluation was terminated early (perhaps by a TOP operator).

If a view does not receive sufficient feedback to guard against it becoming stale, we have two options, either force an update or run (at low priority) artificially generated and specifically tailored *guard queries* just to get feedback. Details related to guard queries are omitted due to space limitations.

5. RELATED WORK

To the best of our knowledge, ours is the first paper to describe a practical method for sampling-based cardinality estimation for complex relational query expressions (e.g. group-by over join). Neither are we aware of any other proposal to exploit query feedback and statistical process control techniques to implement a demand-driven maintenance strategy for samples.

Related work falls into two categories: techniques for improving cardinality estimation in general and using sampling in databases specifically. We briefly outline both categories.

Improving cardinality estimation

Improving cardinality estimation has been an active area of research for many years. Many different techniques have been proposed, especially for dealing with predicates over correlated columns.

One research direction is focused on replacing single-column histograms by alternative techniques better able to handle correlated columns. The CORDS project investigates the benefits of column group statistics using variations of multidimensional histograms [18], while other researcher have proposed using other techniques such as wavelets for cardinality estimation [21]. However, this approach is only helpful for selectivity estimation of predicates defined on a single table. It does not extend to multi-table subexpressions and therefore we do not consider this approach further.

Query feedback can be exploited in different ways to improve cardinality estimation. One extreme–progressive query optimization, e.g. [20]–monitors the number of rows produced by an operator and compares it to the number of estimated tuples. If a certain condition is satisfied, the query undergoes a (partial) re-optimization. Drawbacks of this approach include the potentially high cost of reoptimizations and the limited potential for online reoptimization of a query without losing too much work.

The LEO project uses query feedback in a different way [19]. Predicates together with actual cardinalities collected during execution are stored in a query feedback warehouse. During query optimization, the selectivity of a predicate is computed by looking up actual cardinalities for parts of the predicate and combining the available evidence into an estimate by applying the Maximum-Entropy Principle. This approach may yield very accurate estimates if enough and reliable evidence is available. However, the approach has several drawbacks: the maximum entropy computation may be expensive; if feedback from different points in time is combined, the estimates may be highly unreliable; and the problem of detecting and purging stale feedback remains unsolved.

Our approach builds on the statistics-on-views approach described in [11]. A statistics-only-view is a materialized view that stores only statistics (histograms, in practice) computed from the view result but not the actual result. If a matching subexpression is found during optimization, a new estimate is computed using the statistics from the view and replaces the original estimate. Unfortunately, all statistics must be computed at view creation time and cannot be incrementally generated on demand when queries flow in. It is obvious that sample views can also be used for computing statistics and, furthermore, the statistics can be computed on demand from the sample.

Using sampling

Sampling in database systems [23] is gaining more and more attention both for approximate query answering and internally for query optimization.

An online sample is created for a specific task, for example, histogram creation, and released afterwards. In practice, online sampling is usually limited to block-level sampling over single tables (bi-level sampling: [14]). As shown in [10] sampling cannot in general be pushed below a join, making online sampling over more complex expressions prohibitively expensive. Oracle's Dynamic Sampling Mechanism computes cardinality estimates using online sampling. It can only be used for single-table predicates and relies on block-level sampling. The user has to explicitly trigger the mechanism using the *dynamic_sampling()* hint. The limitation to single-table predicates severely limits the usefulness of this feature.

Offline samples, on the other hand, provide materialized evidence and can be collected on individual tables [3] or on the result of any query expression. Offline samples are either used directly for the application, e.g. Aqua project [2], or exploited to compute histograms or other types of synopses [9]. There is a large body of research on maintenance techniques for offline samples, for example, [24] (for Bernoulli sampling schemes) or [12] (for reservoir sampling schemes). Unfortunately, all techniques implicitly assume an immediate update scheme; update policies driven by statistical requirements of dependent sample synopses have not been investigated.

In this paper we do not propose any new cardinality estimators but focus on the infrastructure needed to exploit existing sample-based estimators in query optimization. There is a vast statistical literature related to estimators. We refer to [26] for a discussion of point estimators, e.g. Wilson [29], and to [4] for a review of confidence intervals for proportion estimates. To parameterize our sequential sampling infrastructure, we benefitted from the work by Haas in [17] and [13]. For distinct-value estimation, we did not propose a specific estimator but showed how to compute, in a sequential fashion, the inputs required to apply a number of well-known estimators, for example, Shlosser [27], Chao and Chao-Lee [6, 7], Poisson-based estimators [30, 25], and jackknife estimators [16, 8]). [15] and [5] provide comprehensive overviews of applicable distinct-value estimators.

6. CONCLUDING REMARKS

In this paper, we introduced sample views as a means to augment traditional cardinality estimation techniques. We outlined how regular view matching finds applicable sample views, described generation of probe queries for selectivity estimation and distinct value estimation by sequential sampling and introduced the concept of quality assurance to determine when to refresh sample views. Based on feedback information gathered during query execution, we apply statistical process control techniques to implement a demand-driven refresh policy. In summary, this paper presents a practical approach to improving cardinality estimation very significantly.

This paper reports initial results on sample views. There are several promising directions for future work. So far we have only used sample views for computing point estimates but other statistics, such as histograms, can also be com-

puted from sample views. Sample views can also be exploited for approximate query answering. Finding ways to exploit query feedback further, for example, to automatically detect what sample views to create or discard, also appears feasible.

7. REFERENCES

[1] M. Abramowitz and I. Stegun. *Handbook of Mathematical Functions*. Dover Publications, 1966.

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. In *SIGMOD*, pages 574–576, 1999.

[3] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *SIGMOD*, pages 275–286, 1999.

[4] L. D. Brown, T. Cai, and A. DasGupta. Interval estimation for a binomial proportion. *Statistical Science*, 16:101–133, 2001.

[5] J. Bunge and M. Fitzpatrick. Estimating the number of species: A review. *Journal of the American Statistical Association*, 88:364–373, 1993.

[6] A. Chao. Nonparametric estimation of the number of classes in a population. *Scandinavian Journal of Statistics, Theory and Applications*, 11:265–270, 1984.

[7] A. Chao and S.-M. Lee. Estimating the number of classes via sample coverage. *Journal of the American Statistical Association*, 87:210–217, 1990.

[8] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, pages 268–279, 2000.

[9] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: how much is enough? In *SIGMOD*, pages 436–447, 1998.

[10] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *SIGMOD*, pages 263–274, 1999.

[11] C. A. Galindo-Legaria, M. Joshi, F. Waas, and M.-C. Wu. Statistics on views. In *VLDB*, pages 952–962, 2003.

[12] R. Gemulla, W. Lehner, and P. J. Haas. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *VLDB*, pages 595–606, 2006.

[13] P. J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *SSDBM*, pages 51–63, 1997.

[14] P. J. Haas and C. Koenig. A bi-level Bernoulli scheme for database sampling. In *SIGMOD*, pages 275–286, 2004.

[15] P. J. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB*, pages 311–322, 1995.

[16] P. J. Haas and L. Stokes. Estimating the number of classes in a finite population. *Journal of the American Statistical Association*, 93:1475–1587, 1998.

[17] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *SIGMOD*, pages 341–350, 1992.

[18] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[19] V. Markl and G. M. Lohman. Learning table access cardinalities with LEO. In *SIGMOD*, page 613, 2002.

[20] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[21] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *SIGMOD*, pages 448–459, 1998.

[22] N.N. *Engineering Statistics Handbook*. National Institute of Standards and Technology, http://www.itl.nist.gov/div898/handbook, 2006.

[23] F. Olken and D. Rotem. Random sampling from database files: A survey. In *SSDBM*, pages 92–111, 1990.

[24] F. Olken and D. Rotem. Maintenance of Materialized Views of Sampling Queries. In *ICDE*, pages 632–641, 1992.

[25] J. K. Ord and G. A. Whitmore. The Poisson-inverse Gaussian distribution as a model for species abundance. *Communications in Statistics*, 15(3), 1986.

[26] J. Sauro and J. R. Lewis. Estimation completion rates from small samples using binomial confidence intervals: Comparison and recommendations. In *Proc. Human and Ergonomics Society 49th Annual Meeting*, pages 2100–2104, 2005.

[27] A. Shlosser. On estimation of the size of the dictionary of a long text on the basis of a sample. *Engineering Cybernetics*, 19:97–102, 1981.

[28] G. B. Wetherill and K. D. Glazebrook. *Sequential methods in statistics, 3rd ed.* Chapman & Hall, 1986.

[29] E. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22:209–212, 1927.

[30] D. Zelterman. Robust estimation in truncated discrete distributions with applications to capture-recapture experiments. *Journal of Statistical Planning and Inference*, 18:225–237, 1988.