Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, Peter Zabback

Exploiting self-monitoring sample views for cardinality estimation

Erstveröffentlichung in / First published in:

SIGMOD/PODS07: International Conference on Management of Data, Beijing 11.06. – 14.06.2007. ACM Digital Library, S. 1073–1075. ISBN 978-1-59593-686-8

DOI: https://doi.org/10.1145/1247480.1247610

Exploiting Self-Monitoring Sample Views for Cardinality Estimation

Per-Ake Larson
Microsoft Research
palarson@microsoft.com

Wolfgang Lehner [*]
Dresden Univ. of Technology
wolfgang.lehner@tu-dresden.de

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

Peter Zabback
Microsoft
pzabback@microsoft.com

ABSTRACT

Good cardinality estimates are critical for generating good execution plans during query optimization. Complex predicates, correlations between columns, and user-defined functions are extremely hard to handle when using the traditional histogram approach. This demo illustrates the use of *sample views* for cardinality estimations as prototyped in Microsoft SQL Server. We show the creation of sample views, discuss how they are exploited during query optimization, and explain their potential effect on query plans. In addition, we also show our implementation of maintenance policies using statistical quality control techniques based on query feedback.

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

Cardinality estimation for query optimization in commercially available database systems relies on statistics, primarily single-column histograms, extracted from the data. Although histograms are useful for a wide range of queries, they conceptually fail in many situations, e.g., in the presence of correlations in the underlying data set, when there are references of user-defined functions in predicates, or when predicate expressions are simply too complex. The use of sampling in general and their exploitation for cardinality estimation has been studied from a conceptual point of view [4]. However, we are not aware of any *practical implementation* of sampling to support cardinality estimation in combination with a statistically sound refresh strategy.

The demo presents our approach to exploiting sample views for cardinality estimation. Sample views are views that materialize only a random sample of the rows produced by the view expression. Figure 1 gives an overview of our approach.

During optimization of a query, (sub)expressions of the query will be matched with existing sample views. After a successful match, the optimizer generates a specifically tailored probe query. This probe query is executed against the sample view and returns summary data needed as input to a statistical estimator. The resulting cardinality estimate is then injected into the optimization process and used in the remainder of this process. The cycle of exploiting sample views, generating cardinality estimates and injecting them is illustrated in the left loop of Figure 1.

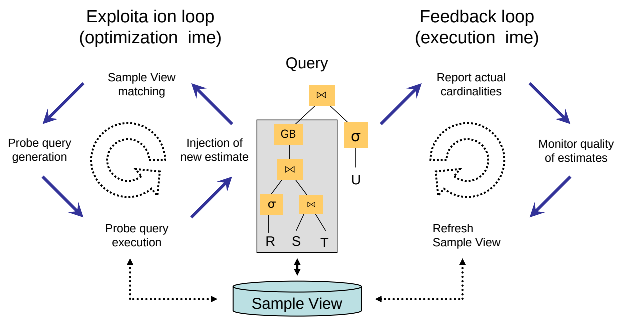


Figure 1: Conceptual Overview of Our Approach

The second (right) loop in the figure illustrates our approach to maintenance of sample views, i.e., we trigger a refresh of a sample view when a feedback monitoring system detects that the sample is no longer statistically valid. Query feedback is collected during runtime; every query that used a sample view during the optimization phase reports the actual cardinality. Errors are normalized relative to the accuracy of the estimator used, resulting in a stream of normalized errors. Every sample view maintains an exponentially weighted moving average of its errors. If the average error exceeds a control bound, with high probability the data has changed so much that the current sample is no longer a valid random sample [3]. The system schedules a background job to recompute the sample.

In summary, our approach not only covers the exploitation part using materialized view matching techniques and probe queries issued during query optimization, it also applies statistical quality control techniques to trigger a sample refresh only when it is statistically justified by query feedback.

[*]Work performed while a visiting researcher at Microsoft.

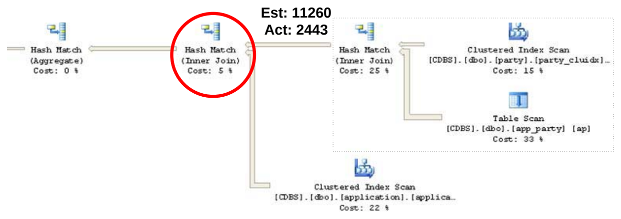
Final edited form was published in "SIGMOD/PODS07: International Conference on Management of Data. Be jing 2007", S. 1073–1075, ISBN 978-1-59593-686-8
https://doi.org/10.1145/1247480.1247610

2. DEMO OF EXPLOITATION LOOP

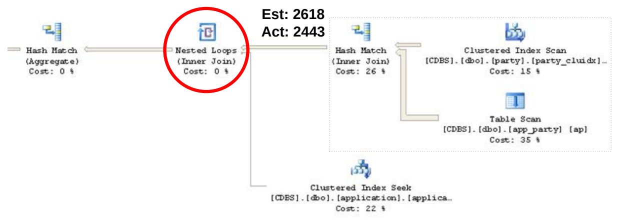
The first part of the demo shows the benefits and overhead of exploiting sample views. In this demo, we use the FCC Media Bureau CDBS Public Database [1]. It is the relational database used by the Media Bureau to process broadcast applications. It is no surprise that various correlations exist among different tables.

For example, the following query summarizes "AM" applications in Hawaii. It contains a join between two correlated tables `party` and `app_party`.

```
SELECT p.party_city, count(*)
FROM party p, app_party ap, application a
WHERE p.party_id = ap.party_id
  AND ap.application_id = a.application_id
  AND a.app_service = 'AM' AND p.party_state = 'HI'
GROUP BY p.party_city
```


Est: 11260
Act: 2443
Hash Match (Aggregate) Cost: 0 %
Hash Match (Inner Join) Cost: 5 %
Hash Match (Inner Join) Cost: 25 %
Clustered Index Scan [CDBS].[dbo].[party].[party_cluidx]… Cost: 15 %
Table Scan [CDBS].[dbo].[app_party] [ap] Cost: 33 %
Clustered Index Scan [CDBS].[dbo].[application].[applica… Cost: 22 %

(a) Original Plan


Est: 2618
Act: 2443
Hash Match (Aggregate) Cost: 0 %
Nested Loops (Inner Join) Cost: 0 %
Hash Match (Inner Join) Cost: 26 %
Clustered Index Scan [CDBS].[dbo].[party].[party_cluidx]… Cost: 15 %
Table Scan [CDBS].[dbo].[app_party] [ap] Cost: 35 %
Clustered Index Seek [CDBS].[dbo].[application].[applica… Cost: 22 %

(b) Better Plan

Figure 2: Query Execution Plans

Since the optimizer has no knowledge of the correlations and always assumes independence, the histogram-based estimate is 11,260 rows output from the join. As shown in Figure 2 (a), the final plan consists of two hash joins. Unfortunately, the real cardinality after the first join is only 2,443 so the chosen plan is suboptimal.

We create a sample view `sv_app_party` over the join between `party` and `app_party`. The view samples and materializes 1% of the join result. Its SQL definition is shown below.

```
CREATE VIEW sv_app_party WITH SCHEMABINDING AS
SELECT p.party_id, p.party_state, p.party_city,
       ap.application_id, ap.party_type
FROM dbo.party p, dbo.app_party ap
WHERE p.party_id = ap.party_id

CREATE UNIQUE CLUSTERED INDEX sv_clidx ON sv_app_party
ON (party_id, application_id, party_type)
ROWSAMPLE 1 PERCENT
```

With the sample view mechanism enabled, the system issues a probe query during optimization and computes the more accurate estimate of 2,618 rows. This results in the plan change shown in Figure 2 (b), where the hash join is replaced by a nested-loop join.

To reduce the cost of probe queries we store the rows of the sample in random order and apply sequential sampling, that is, we read only as many sample rows as is required to produce an estimate of desired accuracy. The technical details can be found in our full paper [2]. The demo shows the probe queries used and how the number of rows consumed by a specific query depends on the predicate.

3. DEMO OF MAINTENANCE LOOP

As outlined in the introductory section, we apply a monitoring loop in order to keep sample views in sync with the underlying data. Standard incremental view maintenance can be applied to sample views but we deemed it to be too expensive. Instead we implemented a quality-based refresh mechanism that fulfills two basic requirements. First, if the base data does not change, it triggers very few 'false alarms,' i.e. it does not trigger unnecessary refresh activity because of normal statistical errors in the estimates. Second, if the base data does change, it reacts as soon as the changes are statistically significant.

To demonstrate both situations, we have developed a monitoring application as shown in Figure 3. In the demo, the application will issue queries against a table with 100,000 rows consisting of only two columns: an `ID` column holds a unique integer value in the range from 1 to 100,000, and the values of the second column are either `0` or `1`, with a probability that can be controlled using the slider at the bottom of the application window.

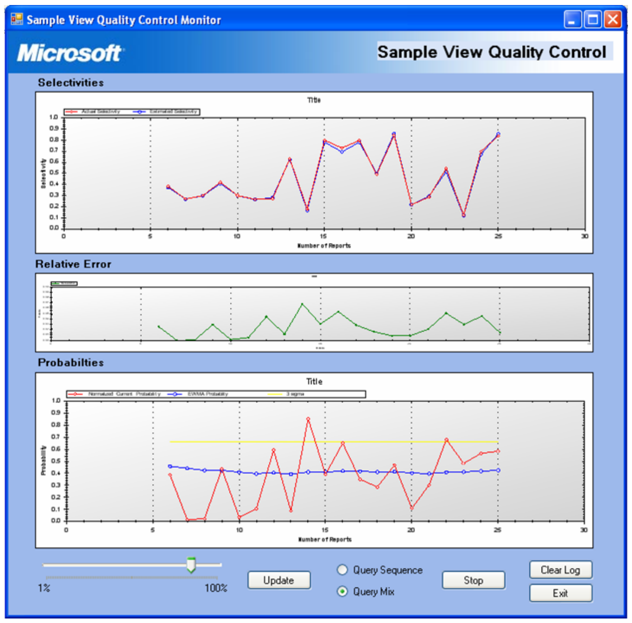
Query Mix Mode

The "query mix mode" of the demo application issues queries against the database system with a range predicate on the `ID` column. The size of the range is randomly determined for every query. As described before, the corresponding sample view is detected and then, a probe query is generated and executed. After the query has finished, the monitoring application shows three different data sets.

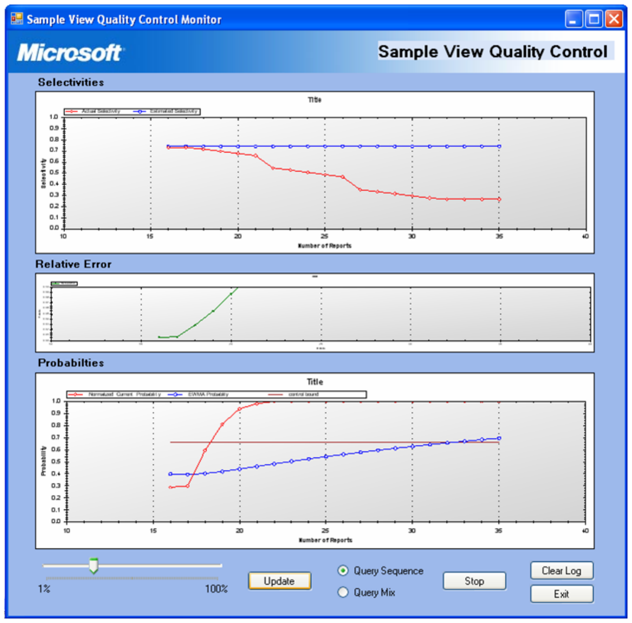
- **Selectivities:** The top sequence window shows the estimated selectivity using the sample view and compares it with the actual selectivity reported by the query.
- **Relative Error:** The middle sequence window plots the relative error of the estimate.
- **Normalized Error:** As outlined in the introduction, we perform a statistical normalization of the estimation error [2]. A value of `0` denotes that the estimate exactly corresponds to the actual value; a value of `1` obviously denotes the maximum error after transformation. The statistically expected error can be found at `0.5`. The bottom sequence window then shows the normalized error for each query and the exponentially smoothed average over the sequence of normalized errors.

The visitor sees a running sequence of queries with corresponding estimates showing a small relative and normalized error. As can be seen in the screenshot – although some of the estimates are rather far from the actual value – the smoothed normalized error permanently stays below a given bound. Within this setup, we additionally explain the error normalization process and the control bound is determined.

Final edited form was published in "SIGMOD/PODS07: International Conference on Management of Data. Be jing 2007", S. 1073–1075, ISBN 978-1-59593-686-8
https://doi.org/10.1145/1247480.1247610



(a) Stable Data Set



(b) Changing Data Set

Figure 3: Screenshots of the Monitoring Application

Refresh-Triggering Mode

The second mode of the demo application ("query sequence mode") illustrates the triggering of refresh activity for the underlying sample view. In this setup, the demo application is continuously issuing the same query with a constant predicate addressing the rows with value `1` in the second column. Upon request, the user may start a transaction updating the values in the underlying table to match a desired query selectivity. In order to illustrate the effect more clearly, we run the read transactions in the `READ UNCOMMITTED` isolation level.

The visitor of the demo will observe the behavior shown in Figure 3 (b). The constant query produces statistically accurate estimates as long as the base data has not changed. As soon as the effects of the update transaction become visible through the feedback mechanism, the normalized error goes up to the maximum. As can be seen, the average error follows the trend and hits the error bound after a few queries. An immediately scheduled refresh recomputes the sample. After the new version of the sample has been installed, the probe queries of the issued queries then exploit the new version, thus producing accurate estimates again. Unfortunately, this behavior is not visible in this screenshot but will be seen in the live demo.

4. SUMMARY

The basic idea of sample views is to use regular materialized view matching infrastructure to find sample views that can be used as a source for cardinality estimation. Probe queries are generated and executed against the sample views during the optimization of a query. This estimation process is unaffected by correlations in the data or other obstacles (e.g., user-defined functions) that cause traditional techniques to produce extremely poor estimates.

In the demo, we describe the infrastructure for exploiting sample views and demonstrate the overall benefit of using sample views for cardinality estimation. We show resulting plan changes, the SQL and query plans for probe queries, and the overhead caused by optimizing and running probe queries during the optimization phase. In addition, we demonstrate how we exploit query feedback to compute normalized estimation errors and implement statistical quality control of the underlying samples. Using two scenarios, we can demonstrate that we do not produce any false alarms if the base data does not change. We can also show how changes in the base data are reflected in the error stream and trigger a refresh of the underlying sample.

5. REFERENCES

[1] Consolidated Database System (CDBS) by the FCC Media Bureau, available at http://www.fcc.gov/mb/databases/cdbs/, 2006.

[2] P. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality estimation using sample views with quality assurance. In *Sigmod*, 2007.

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

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