Statistics on Views

César A. Galindo-Legaria Milind M. Joshi Florian Waas Ming-Chuan Wu

Microsoft Corp.
One Microsoft Way
Redmond, WA 98052, U.S.A.

Abstract

The quality of execution plans generated by a query optimizer is tied to the accuracy of its cardinality estimation. Errors in estimation lead to poor performance, erratic behavior, and user frustration. Traditionally, the optimizer is restricted to use only statistics on base table columns and derive estimates bottom-up. This approach has shortcomings with dealing with complex queries, and with rich languages such as SQL: Errors grow as estimation is done on top of estimation, and some constructs are simply not handled.

In this paper we describe the creation and utilization of *statistics on views* in SQL Server, which provides the optimizer with statistical information on the result of scalar or relational expressions. It opens a new dimension on the data available for cardinality estimation and enables arbitrary correction. We describe the implementation of this feature in the optimizer architecture, and show its impact on the quality of plans generated through a number of examples.

1 Introduction

Cost-based optimization is fundamental to support declarative database query languages efficiently. Application writers need not be concerned with efficient execution algorithms but need only describe logical operations on the abstraction of data stored in tables. It is the job of the query processor to determine and execute the best plan for a query, considering data distributions and physical access paths.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

To select an execution plan for a query, the optimizer enumerates a collection of candidate plans, and picks the one with the least anticipated execution cost. An execution plan is composed of a number of steps of data processing, whose cost is derived based on the particular execution algorithm to use, and the estimated number of rows to process. The quality of plans generated by the optimizer is tied to the accuracy of its cost estimation. Incorrect estimation may lead the optimizer to regard some plans as efficient, when in reality they are very expensive to execute. As effective optimization and good physical design can introduce dramatic performance improvements, so selecting the wrong execution plan can lead to dramatic slowdowns.

Estimating the size of result sets, commonly called the cardinality estimation problem, has been studied for over two decades, and a standard body of techniques has been established and is widely used. The standard approach is to capture the distribution of column values, typically using some form of histograms, which are then used to estimate the number of rows qualifying operations, for example a filter predicate. This approach has shortcomings when dealing with complex queries, and with the full extent of languages such as SQL. Errors grow as estimation is done on top of estimation, so that after several filters and joins, the estimated cardinality may be way off the actual. In addition, there are constructs that simply cannot be estimated from statistics of base table columns. The standard approach when such constructs are encountered is to use a "guess" or "magic number," such as the well known 1/3 data reduction factor for inequality comparisons and 1/10 data reduction factor for equality.

The problem with inaccurate estimation is not only performance, but the introduction of erratic behavior. Queries are "unstable" when there are estimation errors in their optimization. The reason is switch-over points. For example, it is known that index lookup is efficient when there are few rows to lookup, and table scan is preferable if we expect to access all rows; for cases in between, we need to choose, and there is a switch-over point between the two alternatives. Similar switch-over points occur in join order, and in selection of various execution algorithms. When a query is close to a switch-over point, minor changes can make the

optimizer chose one or the other option, which is fine as long as the estimation is correct. However, if a query is *incorrectly* estimated close to a switch-over point, the plan generated will appear to be randomly chosen, with vastly different performance. As observed by the user, adding or removing a simple, non-selective condition in a query may result in dramatic execution slowdown; or a query that used to run fine may suddenly become very slow, after some rows are added to one of the base tables. Two queries that are very close in form and semantics may perform very differently, if one of them happens to use a construct that is not supported by the cardinality estimation model. Software upgrades, and even re-computation of statistics can introduce unpredictable changes in query plans and performance degradation. Such behavior confuses and frustrates developers and DBAs. The system fails to deliver on the goal of high-level, declarative database languages.

To address the shortcomings of the standard, compositional approach to cardinality estimation, we describe in this paper the creation and utilization of *statistics on views* in Microsoft SQL Server, which provide the optimizer with statistical information on the result of scalar or relational expressions. Instead of having base table information as the only source to derive statistics and size of intermediate results, the optimizer has access to pre-derived, accurate statistics on complex expressions. The improved accuracy of estimation increases the quality and reliability of the query processor.

The idea is related to that of materialized views. In the case of materialized views, the system pre-computes and stores the result of some computation, and uses it to speed up query execution when it matches all or part of a user query. For cardinality estimation purposes, we are not interested in the actual view result, but on the *statistical information about the result*. Existing infrastructure for view matching is leveraged and suitably modified to associate such information with (pieces of) a user query.

Providing the optimizer with statistics on views enables arbitrary accuracy on the cardinality estimate of any step in a query execution plan (modulo the capabilities of the view matching service). The optimizer is no longer constrained to derive estimates, through multiple steps, starting from base table statistics, but it has now access to higher-quality statistical information for sub-expressions. The basic statistics technology remains the same, single or multi-dimensional histograms, number of duplicate values, number of rows in a result; but view statistics open a new dimension on the items for which this information is provided.

The paper is organized as follows. Section 2 describes a number of scenarios that are poorly supported by the traditional cardinality estimation approach. Section 3 discusses the principles and requirements for view statistics. Section 4 describes the implementation in SQL Server. In Section 5 we present examples of statistics on views. Section 6 concludes the paper.

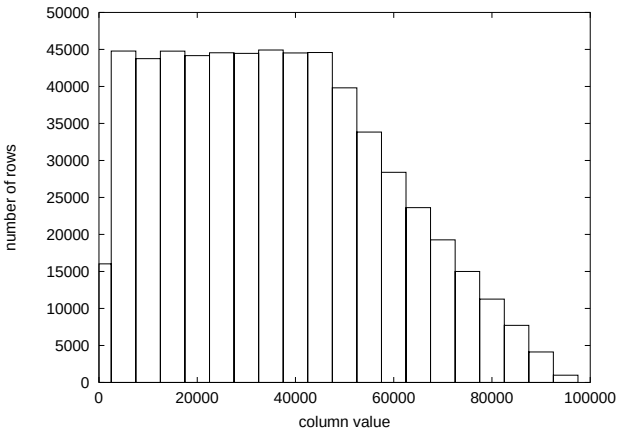


Figure 1: Histogram for column L_EXTENDEDPRICE

2 Statistics on base table columns and its limitations

In the following examples we use tables from the well known, straightforward TPCH schema [9]. Figure 1 shows a histogram that captures the distribution of values for column L_EXTENDEDPRICE from table LINEITEM. The database scale is 100MB, so the table contains 600,000 rows. Values for L_EXTENDEDPRICE range from $1,000 to $96,000, with a uniform distribution on the lower half of the values, and decreasing frequency on the upper half of the values. Such histogram can be used to estimate the qualifying rows in the following query to be about 5,000:

```
Q1:
SELECT * FROM LINEITEM
WHERE
    L_EXTENDEDPRICE>90000
```

To estimate join predicates, the histograms of the columns involved are paired up, and assumptions are made on the degree of matching within each bucket to derive a result. In the special case of foreign-key joins, the exact selectivity of the join predicate is known. For queries containing multiple filter conditions, the usual approach is to estimate the selectivity of each condition separately, and combine the results assuming statistical independence of the filters.

In addition to column histograms, other statistics that systems typically collect are the number of duplicates of a column, or set of columns, and the total number of rows in the table.

We describe next some specific problems with this conventional approach to cardinality estimation. In each case, we present a simple SQL query that exhibits the difficulty. In isolation, some of the queries will actually perform fine, as it is only the *final* result size that is incorrectly estimated. But when the problem described is a piece within a larger query, the incorrect cardinality estimation will be used as the basis for costing later operations in the execution plan,

leading to problems in plan selection, erratic behavior and poor performance.

Predicate involving scalar expressions and/or multiple columns of the same table. For example, the following query is a simple modification of our earlier query Q1. It finds items whose *discounted* price is over $90000:

```
Q2:
SELECT * FROM LINEITEM
WHERE
   L_EXTENDEDPRICE*(1-L_DISCOUNT)>90000
```

The discounted price is computed in the query using simple arithmetic operations over two columns of the table. Estimating the distribution of the result based on the column statistics, while conceivable, can easily introduce large errors, and it is unfeasible in the general case. However, from a user's perspective, this is a very intuitive and simple condition to use. SQL allows a number of scalar operations whose properties are hard to model in cardinality estimation. For example, arithmetic modulo (%), CASE-WHEN-ELSE-END (similar to conditional evaluation in C (pred ? value1 : value2)), and string operations such as concatenation and substring.

Violation of independence assumption. For example, the following query retrieves the customers for a particular nation:

```
Q3:
SELECT * FROM CUSTOMER, NATION
WHERE C_NATIONKEY = N_NATIONKEY
AND N_NAME = 'BRAZIL'
```

Assuming a foreign key relationship between the tables, and 25 countries, each individual condition can be estimated perfectly with selectivities 1/card(nation) and 1/25, respectively. Then, using the standard independence assumption, the composite selectivity is computed as the product of the two. However, it is possible to have extreme skew in the number of customers per country. If customers in Brazil are either very few or very many, in comparison with the average for all countries, then the estimated result size will be arbitrarily bad.

This is a typical scenario in star schemas, where a large fact table is connected to a number of smaller dimension tables. Some values of a given dimension may be much more frequent than others in the fact table, and the standard independence assumption will lead to incorrect estimates.

Another example of violation of independence occurs when estimating the result of duplicate elimination on a number of columns, when the number of distinct rows may be much smaller than the product of distincting each of them, due to data correlation.

The assumption of independence is pervasive throughout the derivation of cardinality estimates. Unlike the earlier case, where the optimizer resorts to a guess and therefore recognizes the high likelihood of an estimation error, the error introduced by non-independent conditions goes undetected at optimization time.

Aggregate results. For example, the following query retrieves orders whose total discounted amount is over $400,000:

```
Q4:
SELECT L_ORDERKEY FROM LINEITEM
GROUP BY L_ORDERKEY
HAVING 400000 <
   SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT))
```

These queries require the estimation of both the number of groups, as well as the distribution of the aggregate result, in this case the computed sum. For some aggregates, it is possible to formulate an approximate result distribution, but a large error can easily be introduced.

"Advanced" SQL operations. For this example we do not use the TPCH schema because we need a recursive relationship. The following query uses a table expression MgrTransEmp that computes the transitive closure of the manager-report relationship from table EMP(empid, mgrid). It retrieves all transitive reports of a particular manager. ANSI syntax is used for the transitive closure operation.

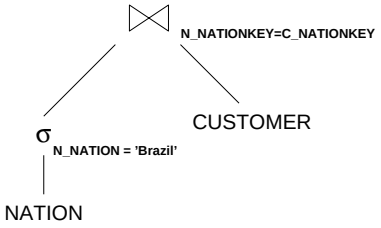
```
Q5:
WITH MGRTRANSEMP(MGR, TR_EMP) AS
   (SELECT EMPID, EMPID FROM EMP
   UNION ALL
   SELECT MGR, EMPID
   FROM MGRTRANSEMP, EMP
   WHERE MGRTRANSEMP.MGR = EMP.MGRID)
SELECT * FROM MGRTRANSEMP
WHERE MGR = 'JOHN SMITH'
```

We are not aware of any work addressing estimation of these complex queries from the standard statistics maintained on table columns. Other "advanced" operations have been added, and will likely continue to be added to SQL. An example is the SQL Statistical Extensions recently proposed to ANSI, which extend the SQL platform to allow computation of summary information like moving averages, percentiles with respect to a group, and ranking of rows. User-defined functions and aggregates present a problem as well.

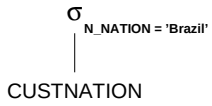
These are all useful constructs of the language for people to write applications. Handling "the common case," and having unpredictable performance when queries contain operations that are difficult to estimate, translates into a problem of quality and reliability of the system in the eyes of users, who do not (and should not need to) understand the shortcomings of the internal implementation and accommodate for them. This problem has to be addressed for SQL to be a truly robust, high-level application platform.

3 Exploiting statistics on views

The following example illustrates the usage of statistics on views during optimization. Consider query Q3 of Section 2. Assume a view CUSTNATION defined as



(a) Original query: Estimate two predicates using independence assumption



(b) Equivalent query using view: Estimate one predicate, no independence assumption

Figure 2: Equivalent alternatives for query

```
CREATE VIEW CUSTNATION AS
SELECT * FROM CUSTOMER, NATION
WHERE C_NATIONKEY = N_NATIONKEY
```

Figure 2 shows equivalent operator trees, the original query Q3 and a rewritten form using CUSTNATION. The conventional cardinality estimation approach would use 2 (a). It proceeds by estimating both predicates N_NATION = 'BRAZIL', and C_NATIONKEY = N_NATIONKEY, and then combining the result assuming statistical independence, which in this case introduces a significant error in the estimate. We can make use of the statistics on CUSTNATION in order to estimate the cardinality of the result by using expression 2 (b). In this form there is only one predicate to consider, N_NATION = 'BRAZIL'. The independence assumption is no longer needed, and the result size can be estimated accurately from the statistics on CUSTNATION.N_NATION.[1]

Just as is the case with standard materialized views, and with query optimization in general, equivalence of expressions plays a key role. Given the equivalence, if view CUSTNATION were materialized the optimizer could use either of the two forms for execution, and would pick the one with least anticipated cost. For the same reason, the optimizer can use either of them to estimate the size of the result, and should pick the one with least anticipated estimation error.

Picking an alternative to estimate the size and statistics of the result is independent of the plan selected for execution. In fact, it will often be the case that the view, say CUSTNATION, is not materialized but is used only to provide accurate statistical information about the result. The optimizer in this case maintains a number of logically equivalent alternatives that are not "implementable," but only contribute metadata to the optimizer.

[1]The general idea here is the same as that of Statistics on Intermediate Tables (SIT) from [1]. The approach described in that paper targets join queries and presents a mechanism with minimal assumptions on the optimizer architecture, based on intercepting calls to the cardinality estimation functions. In a product, we prefer a tight integration of the mechanism within the optimizer framework, leveraging functionality of existing components, and yielding what we consider a more efficient, maintainable, and robust architecture.

To exploit statistics on views we can leverage existing system infrastructure built to support materialized views. However, there are a number of requirements that differ from the traditional materialized view context.

- View matching must be integrated with cost-based plan enumeration. For standard materialized views, it is possible to implement view selection using heuristics, in a pre-processing step, and output a chosen rewritten query that is then passed to the cost-based optimizer. However, this approach does not extend to views for cardinality estimation. The cost-based optimizer generates a number of intermediate expressions of interest, as part of enumerating feasible plans for the query. It is on these dynamically generated expressions that we require cardinality estimation, and would like to identify views that can provide information for increased accuracy.

- The view matching algorithm needs to scale to hundreds of views. Work on this front has been done in [3], which we use in our implementation. A few carefully selected materialized views may be sufficient to achieve dramatic performance improvements in the execution of queries. However, views used for statistics have much smaller storage requirements, since the view result is not required. They have a much higher ceiling in terms of the number of useful views one may want to have.

- We need to extend the class of queries handled by the matching algorithm. In the context of traditional materialized views, maximum benefit is obtained when the view stores a "small" result obtained by an "expensive" computation, as it is the case with aggregates. Thus, for materialized views, it may be adequate to limit support to a subclass of common operations where view substitution has a large query execution payoff. For the purpose of cardinality estimation, we want to allow *any* view to participate in matching, especially those with complex operations that are beyond what can be easily derived from statistics on base table columns.

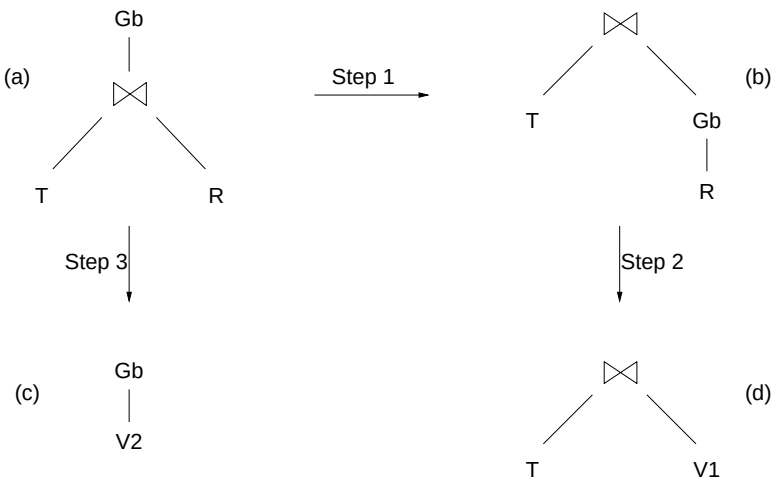


Figure 3: A group of expressions equivalent under different optimization steps

- The column coverage test in view substitution can be relaxed. To use a materialize view in query execution it must provide all columns needed for the query (or extra joins need to be added to retrieve the missing columns, as it is done when dealing with non-covering indices). If only statistics are used, the optimizer can make use of metadata about result size or statistics on some columns, even if the view does not provide information for all columns.

It is worth pointing out that, although the view matching problem on Select/Project/Join is NP-hard [7], instances found in practice are far simpler than the worst case. Checking equivalence (or subsumption) between join queries corresponds to checking graph isomorphism, where nodes correspond to tables and predicates correspond to edges. If the *the same table* is joined multiple times (i. e. self joins), then these are graphs with unlabeled nodes; each usage of the table in one query could correspond to any other usage in the other query, and multiple correspondences need to be considered, thus leading to the high complexity of the problem. However, if no self-joins are involved, then table names or ids make for unique node labels, making only one node correspondence feasible for the graph. Though theoretically possible, customer queries on normalized schemas do not make extensive use of self-joins. This removes a source of complexity and enables fast view matching.

4 Implementation in SQL Server

To meet the requirements outlined in the previous section, we leverage existing query processor technology implemented in Microsoft SQL Server. The usage of statistics on views relies on other components already present in the query optimizer, in particular exploration and management of alternatives, and *view matching*, so we start by briefly reviewing those components. We assume the reader familiar with the basic principles of transformation-based optimization (see e.g. in [6, 2]).

4.1 Algebraic optimization

The algebraic query optimizer of SQL Server is based on the Cascades optimizer architecture [4]. Some of its key features are:

- The optimizer uses algebraic transformation rules to implement exploration of the search space. Each transformation takes an input operator tree and produces one or more equivalent alternatives.
- Optimization is strictly cost-based, i.e. queries are not rewritten on the basis of heuristics; rather, all alternatives are costed and eventually the plan that is optimal according to the cost model is chosen.
- The architecture enables extensibility, in the sense that new transformation rules can be incorporated to the system to generate additional equivalent expressions without disturbing existing optimizations or the cost-based selection mechanism.

We implement view matching in the optimizer by adding a transformation rule. This rule encapsulates the process of matching a relational operator tree with a set of candidate views; in case one or more views match or subsume the original expression, the transformation returns equivalent expressions based on these views. This scheme is explained in some detail in [3].

To see how different optimizations interact with view matching, consider the example shown in Figure 3. Figure 3(a) is the initial expression passed into the optimizer. It consists of a join between tables T and R and an aggregate over the result. In step 1, a transformation pushes the GroupBy operator below join (it of course checks a number of conditions on the join predicate and grouping columns to decide if the transformation is valid). The second and third transformations detect that a sub-expression matches

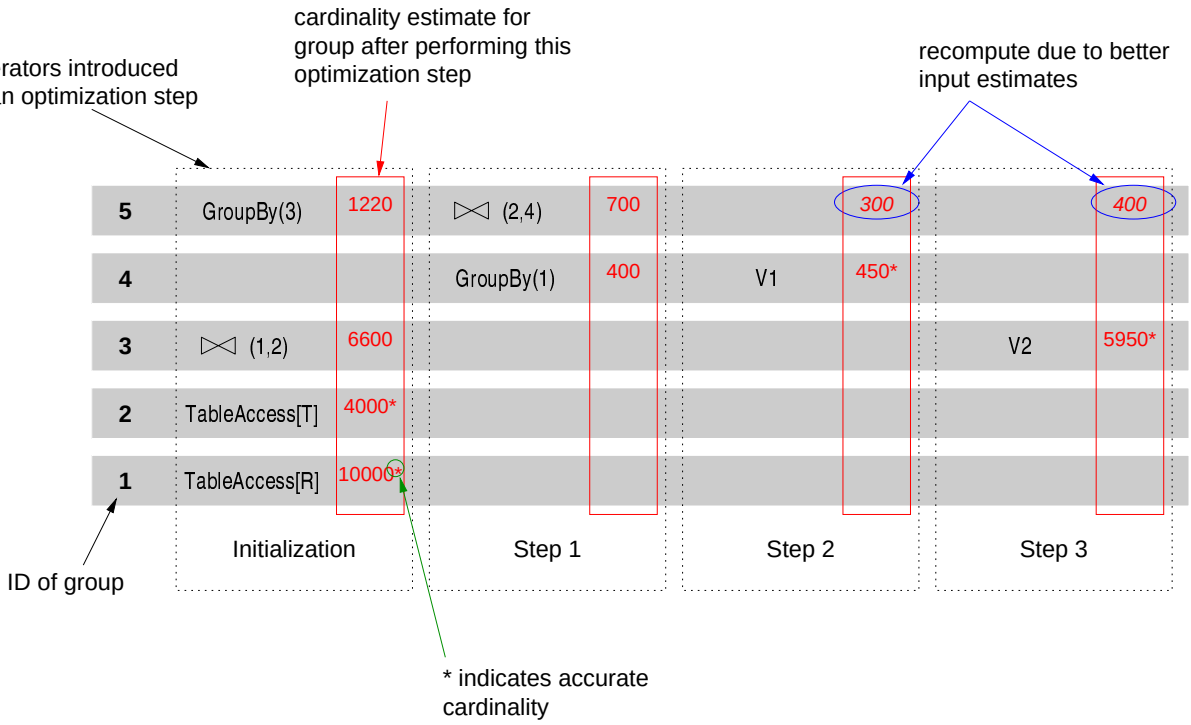


Figure 4: Optimizer look-up table that implements internal encoding as groups. Horizontal boxes depict groups; number in parentheses following operators denote group(s) of inputs. Vertical dashed boxes indicate operators introduced in each optimization step analogous to Fig. 3, numbers in solid boxes show cardinality estimates after each optimization step

an existing materialized view, and outputs a new, equivalent operator tree based on such view.

The use of views is not limited to exact matches, but may make use of residual operations. This was the case in our earlier example, in Figure 2. The view matching rule would be passed directly expression 2(a), and it would return 2(b).

4.2 Management of alternatives

Instead of keeping a collection of separate, fully expanded operator trees, the optimizer uses an efficient encoding that maximizes re-use of common sub-expressions. This encoding is based on a system of *groups*. Each group contains operators that are the root of equivalent sub-expressions. The inputs of each operator are placed in other groups according to the same principle. This compact encoding is similar to the table of sub-plans used by dynamic-programming join enumeration [8], except the population of the table is driven by the application of transformation rules and it handles operators other than joins. For a detailed description of the encoding see [5, 4].

In Figure 4 this encoding is depicted for the alternatives in Figure 3. During initialization, each operator of the original expression is placed into a separate group. The references between operators are substituted by references to groups—indicated by the Groups' IDs in parentheses. For example, once the GroupBy operator is inserted into Group 5 its reference to the join below is turned into a reference to Group 3, since any operator added to Group 3 at a later time is equivalent to the join and can serve as input to the GroupBy. The dashed vertical boxes indicate which operators are introduced by an optimization step. Step 1 adds a new GroupBy as well as a join which go to Groups 4 and 5. In Step 2, view V1 is added to group 4, and finally, view V2 is inserted in Group 3 as result of Step 3.

4.3 Cardinality estimation

The groups of alternative operators form the basis for the cardinality estimation framework. Each operator has an associated function to derive an estimate of output cardinality and statistics, based on the respective information on its inputs. Since all expressions in a group produce the same result, cardinality and statistics for such result can be derived using *any* one of those operators. The quality of estimation will be higher in some operators, lower in others. This is reflected by a measure of their *estimation reliability*, which we use to pick among the multiple alternatives. Estimation reliability is a function of several factors.

- The *quality of statistics* available for the estimation. These may be unmodified source statistics, or statistics propagated and modified through other operators, or it may be that required statistics are not available.

- Reliance on the *independence assumption*. Estimating one predicate has higher expected quality than es-

timating two predicates and then combining using the independence assumption.

- The *type of estimation* to be done. For example, estimating the number of distinct values for the result of GroupBy is in general more difficult than estimating the result of a simple filter condition.

A view or table reference has maximum reliability. When a group includes an operator that is a direct reference to the view, the cardinality will be picked up directly from it and used for the group.

In Figure 4, we show the cardinality estimate for all affected groups after each individual optimization step. Accurate, reliable estimates are indicated by $\star$. The alternative expression inserted by Step 1 comes with higher reliability than the original one due to the type of estimation. After Step 2 we obtain an accurate value for the cardinality of Group 4, which affects also the cardinality of Group 5. Finally, using view V2 provides even higher reliability, leading to the most accurate estimate for Group 5.

To estimate the result size of a group, we use the operator with highest reliability. Note that this operator may not provide statistics for all columns; for example, in the case of a view that has very high reliability but does not contain all columns from the query. In general, we will use multiple operators in a group to collect statistics on the columns required.

In practice, we do not actually re-compute estimates in later operators as a result of *each* optimization step, as it was shown for illustration purposes in Figure 4. We batch a number of changes and only then do we re-derive estimates for which a more reliable alternative has become available. A dependency constraint on this computation is that accurate estimates be present by the time we start generating and costing physical operators.

5 Examples

In this section we go over a number of examples where the use of statistics on views improves the quality of plans generated. We present relatively simple queries that illustrate the concept and show the impact of the feature; clearly, the usability and effectiveness extends to larger and more complex queries.

We use the 1GB scale of the TPC-H benchmark database, with the standard indexing restrictions (basically, no covering indexes). Table 1 shows a summary of query execution times in seconds when only base table statistics are available to the optimizer and then with statistics on views.

	base table stats only	stat on views
E1	100	90
E2	14.2	3.6
E3	1.1	0.1

Table 1: Execution times in seconds

Next we go over each of the queries, describe the plan changes, and show the commands available to a user for creating statistics on views.

5.1 Example E1

This query retrieves all orders of lineitems with a discounted price less than $900.

```
E1:
SELECT ORDERS.*
FROM LINEITEM, ORDERS
WHERE
    L_EXTENDEDPRICE * (1 - L_DISCOUNT) < 900
    AND L_ORDERKEY = O_ORDERKEY
```

With only base table column statistics, the optimizer is left "guessing" a selectivity for the arithmetic expression. The selectivity of the predicate is underestimated, which makes hash join appear as the preferable join method. The resulting plan is shown in Figure 5a).

To assist optimization in estimating the cardinality for the filter on LINEITEM, we create statistics on views using the following commands:

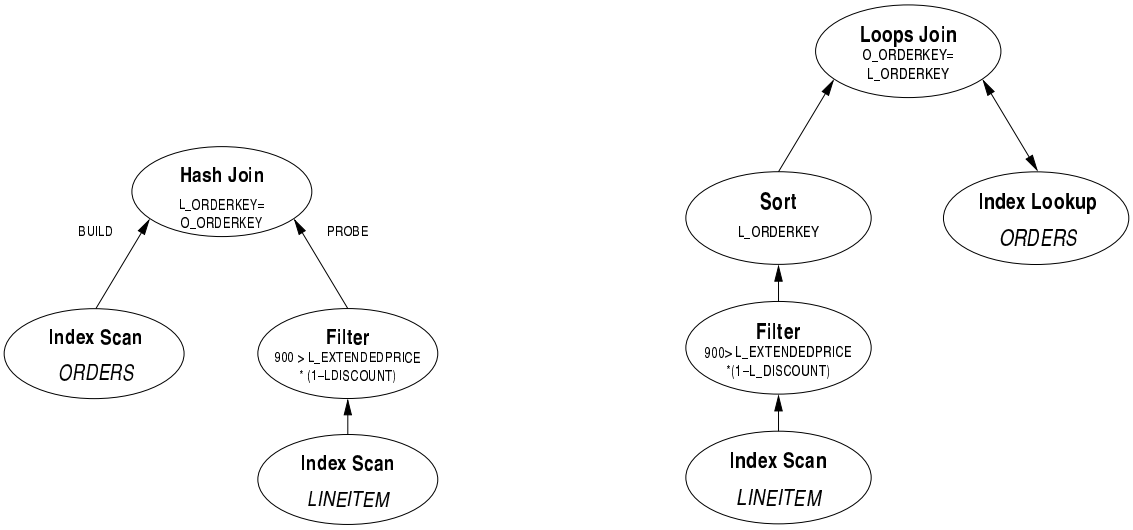
```
V1:
CREATE VIEW V1 AS
SELECT
    L_EXTENDEDPRICE * (1 - L_DISCOUNT)
    AS DISCOUNT_PRICE
FROM LINEITEM


S1:
CREATE STATISTICS S1 ON
    V1(DISCOUNT_PRICE)
```

The syntax for creating statistics on a view result from removing an earlier restriction in the product. Users of SQL Server know they can use CREATE INDEX ON X both for tables and views. Creating an index on a view is the mechanism currently used to compute the result of the view and store it, in an index, effectively creating a materialized view. Users also know that CREATE STATISTICS ON X currently works only for tables. With statistics on view, this asymetry is removed and the syntax is allowed on views also, with a clear semantics that fits within the user's mental model.

Using V1 during optimization enables a more accurate estimate for the arithmetic expression and, subsequently, identifies the index lookup into ORDERS as a more efficient solution. The new plan output by the optimizer is shown in Figure 5b).

5.2 Example E2

Another frequently encountered problem area are conditional statements like CASE-WHEN-ELSE-END. This construct is typically used in views in order to encapsulate certain business logic. Consider the following view defining a rating of customers based on their account balance



(a) (b)

Figure 5: Query plans for example E1 with base table statistics (a) and statistics on views (b)

```
CREATE VIEW RATING AS
SELECT *, R_RATING = CASE
WHEN C_ACCTBAL < 1000 AND
    C_MKSEGMENT='AUTOMOBILE' THEN 'BAD'
WHEN C_ACCTBAL > 10000 AND
    C_MKSEGMENT='MACHINERY' THEN 'GOOD'
ELSE 'UNKNOWN'
END
FROM CUSTOMER
```

Using this view, users could write a query to retrieve information about orders for customers with *good* rating:

```
E2:
SELECT C_CUSTKEY, O_ORDERKEY
FROM RATING, ORDERS
WHERE C_CUSTKEY = O_CUSTKEY
AND R_RATING = 'GOOD'
```

With only base table statistics, the optimizer overestimates the result of the filter and chooses a merge join, leveraging the sort order from the index on ORDERS. The resulting plan is shown in Figure 6a).

In this scenario, it makes sense to create statistics directly over the view that is used to abstract the business logic:

```
S2:
CREATE STATISTICS S2 ON
RATING(R_RATING)
```

With the availability of statistics on RATING, the original estimate is corrected and the optimizer makes a better plan choice, using an index lookup into ORDERS rather than a merge join. Figure 6b) shows this more efficient plan.

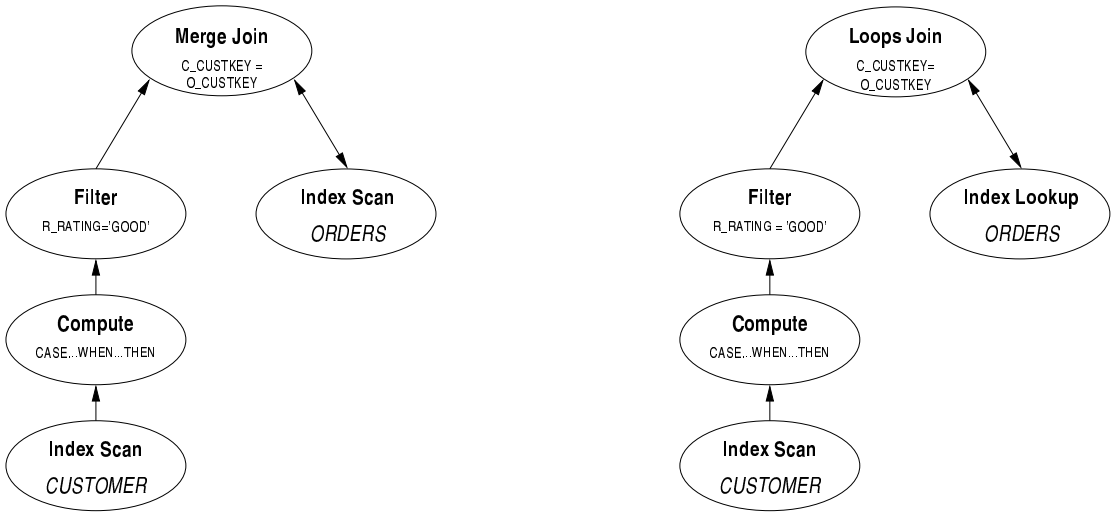
The fact that we use this view by name in the actual query, and that it happened to have relevant statistics, is immaterial for the utilization of statistics. Since exploiting statistics on views is done automatically based on expression matching, the query need not mention explicitly any view to have relevant statistics used.

5.3 Example E3

This example is more complex and it illustrates a number of additional points. It computes the average account balances for customers in a given country. The query uses a subselect that computes the account balances per country, for all customers, and then joins that with nations.

```
E3A:
SELECT *
FROM NATION,
      (SELECT C_NATIONKEY,
            AVG(C_ACCBAL) AS AVGBAL
            FROM CUSTOMER
            GROUP BY C_NATIONKEY)
      AS CUST_SUMMARY
WHERE N_NATIONKEY = C_NATIONKEY
AND N_NAME = 'MEXICO'
```

This case is interesting when the distribution of customers in countries is not uniform, because the selected country has an impact on the number of qualifying customers. Since TPC-H uses a uniform distribution, we modified the data to convey our point. We added a new country



(a) (b)

Figure 6: Query plans for example E2 with base table statistics (a) and statistics on views (b)

to the NATION table, Mexico, and then added ten customers for the new country.

With only base table column statistics, the optimizer assumes the query has to retrieve an "average" number of employees, regardless of the nation selected. It produces the plan shown in Fig. 7a), which uses early aggregation on C_NATIONKEY before joining with NATION (as suggested by the query syntax). However, this requires a full scan of customers, which is inefficient if there are few qualifying customers for the country chosen.

To address this scenario, data skew on join result, we create statistics on a join view:

```
V3:
CREATE VIEW V3 AS
SELECT N_NATIONKEY, N_NAME
FROM NATION, CUSTOMER
WHERE
   N_NATIONKEY = C_NATIONKEY


S3:
CREATE STATISTICS S3 ON
   V3(N_NAME)
```

Availability of these additional statistics affects the plan selection when the selected country is Mexico (with only ten customers, as explained above). The table scan with early aggregation on the customer's nationality is correctly estimated to be substantially more costly than utilizing the index on C_NATIONKEY of CUSTOMER, and executing the aggregation after the join. Since the index is non-covering, it is necessary to "join" back into the main table to retrieve necessary columns not included in the index. Figure 7b) shows the resulting plan in this case.

Optimizing the query for a country with a large number of customers, e.g. Germany, with the new join statistics produces the early aggregation plan, since the number of qualifying customers (accurately estimated) justifies the table scan.

This final example illustrates several points on the behavior of the complete, integrated system:

- By integrating cardinality estimation and view matching in the general plan generation framework, the view can be leveraged despite the fact that it does not match the original input expression. After applying a number of transformation rules the view matching establishes the link between an alternative of the input and the view.
- Statistics on views are effective in providing statistics information even if they do not match the plan that is finally chosen for execution. In case of the 'Germany' query, the statistics on the view are used, and make the optimizer choose a plan that does not contain an expression equivalent to the view.
- The presence of statistics on views does not limit the choice of plans but only assists in the reliable estimation of cardinality and costing.

6 Conclusions

Accurate cardinality estimation has been a long-standing problem for reliable plan selection in cost-based optimiz-

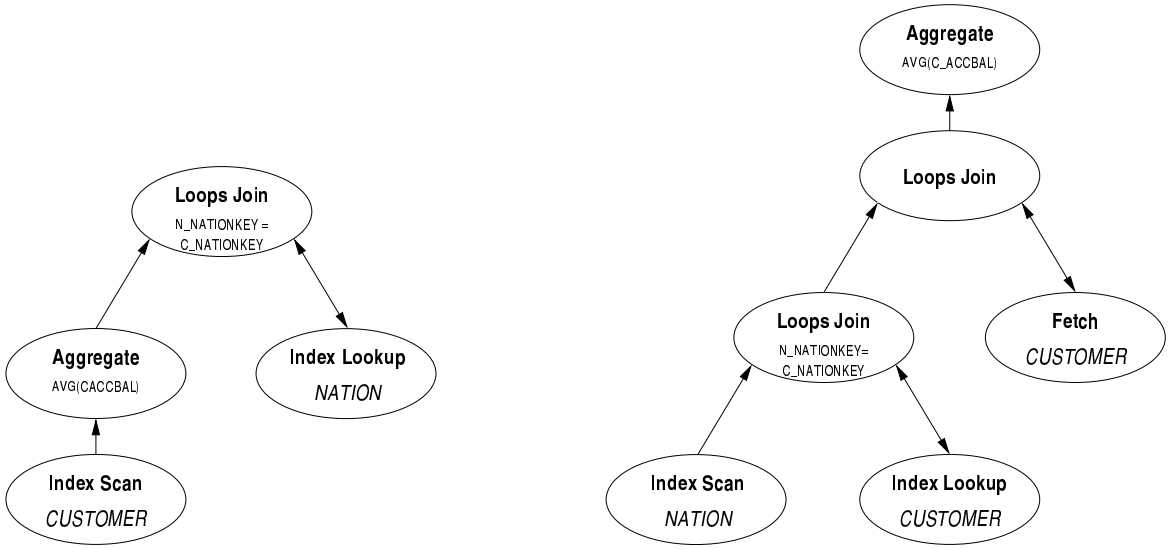


Figure 7: Query plans for example E2 with base table statistics (a) and statistics on views (b)

ers. The conventional approach starts with statistical information from base tables, and based on that estimates result sizes after each data processing step. This approach has fundamental limitations. Errors grow as estimation is done on top of estimation, and some language constructs are hard or impossible to handle, turning "estimation" into a guess. Errors in cardinality estimation translate into severe performance degradations, and poor performance reliability. Depending on the data set and the makeup of the query, "bad plans" can be triggered by changes as simple as creating a new index or adding a few rows to a table.

To address the shortcomings of this conventional approach, we described in this paper *statistics on views* in Microsoft SQL Server, which provide the optimizer with statistical information on the result of scalar or relational expressions. This opens a new, orthogonal dimension on the data available to the optimizer for estimation, and enables arbitrary correction. We showed the impact of this product feature on the quality of plan selection, on a number of simple, specific cases where traditional estimation falls short.

The concept can be explained to users leveraging their understanding of materialized views and statistics, so that it fits within their mental model. The syntax extensions effectively amount to removing an earlier restriction — users knew they could use CREATE INDEX ON X both for tables and views, but could only use CREATE STATISTICS ON X for tables; now the second command works for both types of objects.

Our internal architecture for the feature leverages existing optimizer functionality and re-factors the cardinality estimation task across various orthogonal components, in particular exploration and management of alternatives, and view matching. Enhancements in functionality or performance on each component are immediately reflected in cardinality estimation. For example, handling new operators in the view matching service will benefit both materialized views as well as cardinality estimation; and augmenting the set of transformation rules will extend the space of candidate execution plans as well as the alternatives to derive cardinality estimation.

Considering the difficulties with traditional cardinality estimation encountered on real systems, we see statistics on views as a crucial building block to take quality and reliability of query optimization to the next level.

References

[1] N. Bruno and S. Chaudhuri. Exploiting Statistics on Intermediate Tables for Query Optimization. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Madison, WI, USA, June 2002.

[2] E. Elmasri and S. B. Navathe. *Fundamentals of Database Sytems*. Benjamin/Cummings, Redwood City, CA, USA, 2nd edition, 1994.

[3] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, Santa Barbara, CA, USA, June 2001.

[4] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, Sept. 1995.

[5] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 209–218, Vienna, Austria, Apr. 1993.

[6] H. Korth and A. Silberschatz. *Database Systems Concepts*. McGraw-Hill, Inc., New York, San Francisco, Washington, DC, USA, 1991.

[7] A. Y. Levy, A. O. Medelzon, Y. Sagiv, and D. Strivastava. Answering Queries Using Views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 96–104, San Jose, CA, USA, May 1995.

[8] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 306–315, Athens, Greece, Sept. 1997.

[9] Transaction Processing Performance Council, San Jose, CA, USA. *TPC Benchmark H (Ad-hoc, Decision Support)*, Revision 1.2.1 1999.