StatAdvisor: Recommending Statistical Views

Amr El-Helw Ihab F. Ilyas Calisto Zuzarte

University of Waterloo
{aelhelw, ilyas}@cs.uwaterloo.ca

IBM Toronto Lab
calisto@ca.ibm.com

ABSTRACT

Database statistics are crucial to cost-based optimizers for estimating the execution cost of a query plan. Using traditional basic statistics on base tables requires adopting unrealistic assumptions to estimate the cardinalities of intermediate results, which usually causes large estimation errors that can be several orders of magnitude. Modern commercial database systems support statistical or sample views, which give more accurate statistics on intermediate results and query sub-expressions. While previous research focused on creating and maintaining these advanced statistics, only little effort has been done towards automatically recommending the most beneficial statistical views to construct. In this paper, we present `StatAdvisor`, a system for recommending statistical views for a given SQL workload. The `StatAdvisor` addresses the special characteristics of statistical views with respect to view matching and benefit estimation, and introduces a novel plan-based candidate enumeration method, and a benefit-based analysis to determine the most useful statistical views. We present the basic concepts, architecture, and key features of `StatAdvisor`, and demonstrate its validity and benefits through an extensive experimental study using a prototype that we built in the IBM® DB2® database system as part of the *DB2 Design Advisor* tools.

1. INTRODUCTION

Cost-based query optimizers rely on a cost model to choose the best possible execution plan for a given query. The accuracy of cost estimates is the main factor that affects the quality of the query execution plan. Cost estimates depend mainly on cardinality estimations of various sub-plans (intermediate results) generated during optimization. Traditional query optimizers often assume data uniformity and independence of query predicates to estimate output cardinalities of intermediate result sets, using statistics built over base tables. However, these assumptions are usually incorrect, causing cardinality estimates to be off by orders of magnitude, leading to suboptimal execution plans.

Base table statistics, including histograms, have several shortcomings when dealing with complex query constructs. Examples include predicates with arbitrary expressions on multiple columns and aggregate functions. It is not uncommon to use a *guess* or *magic number* as the selectivity estimate of the predicates that have these constructs [11].

1.1 Statistical Views

The inability to accurately estimate the output cardinality of complex query expressions triggered the idea of collecting statistics on views [11], also known as SITs – statistics on intermediate tables (or SQEs – statistics on query expressions) [6, 7]. In this paper, we use the term *statistical views* (or *statviews*), which can be defined as follows:

Definition 1.1. A statview is a view definition (SQL query) augmented with statistics collected on the result of executing this view, without the actual data.

The statistics that can be collected on a statview are the same as those that can be collected on a base table, e.g., the number of tuples in the statview, the number of distinct values in each column, the highest and lowest values in each column, and optionally, column group statistics, histograms and frequent values on some or all of its columns. Statviews make it possible to estimate the cardinality of some complex sub-expressions that otherwise would have to be guessed or estimated using unrealistic assumptions.

Example 1.1. Consider the following query Q:

```
SELECT * FROM Car, Owner
WHERE Car.OwnerID = Owner.ID
AND Owner.Sal=3000
AND Owner.Age=30
AND Car.Price*(1-Car.Discount)<5000
```

To estimate the output cardinality of this query, the optimizer has to estimate the output cardinality of each table after applying the local predicates, then estimate the output cardinality of the join operator. To estimate the cardinality of the `Owner` table, the optimizer estimates the selectivity of each of the two predicates `Owner.Sal=3000` and `Owner.Age=30` from the number of distinct values (and possibly the frequent values) in the columns `Sal` and `Age`, respectively. The combined selectivity of the two predicates is estimated assuming independence. The independence assumption can be relaxed if a two-dimensional histogram on `Age` and `Sal` is available (in which case, the uniformity assumption is employed to some extent to interpolate values within histogram buckets). However, two-dimensional histograms on arbitrary pairs of attributes are not usually available. Estimating the selectivity of a predicate involving an expression like (`Price*(1-Discount)<5000`) is usually hard using base table statistics, and most optimizers obtain a selectivity estimate for such predicates using some predefined magic number [11]. The estimation errors in both tables are further magnified as a result of the join predicate [13]. On the other hand, assume that we have the following statviews:

v_1: `SELECT * FROM Owner WHERE Sal=3000 AND Age=30`

v_2: `SELECT * FROM Car WHERE Price*(1-Discount)<5000`

Collecting statistics on these statviews gives us accurate information about the number of rows in each view, thus reducing the estimation error considerably.

To the best of our knowledge, little work has been done to automate the process of deciding which statistical views to create [6]. In addition, previous work did not study the interaction of multiple statviews when presented together to the query optimizer. In this paper, we focus on recommending the most beneficial statviews for a workload, taking into account the interaction between statviews and their effect on query plans.

1.2 Background: Statistics Collection

In this section, we briefly explain some of the techniques currently used to collect statistics on views. The simple approach to collect statistics on a statview is to execute the statview's query, materialize the results, collect statistics on the results, then drop the query's results. This approach is not efficient, especially when there are multiple statviews, many of which involve the same tables.

A better approach is to create and maintain random samples of the base tables. Suppose there are two statviews $v_1 = \sigma_{p_1}(A)$ and $v_2 = \sigma_{p_2}(A)$. To collect statistics on these statviews, the sample of table A is scanned only once, and each tuple is checked against the predicates p_1 and p_2.

Unfortunately, base table samples cannot be used for statviews with multiple joined tables, since joining the base table samples does not yield a random sample of the join [2, 8, 18]. To overcome this problem, join synopses [2] can be used. The join synopsis for table A is built as follows [4]:

1. Create a uniform random sample of A.
2. For every table B such that A has a foreign key to B, join the sample of A with the full table B.
3. Repeat Step 2 recursively, i.e., for each table B from Step 2, follow all its foreign keys.

Now suppose there is a statview on a group of tables that are all joined using foreign key joins. Join synopses can be used to collect statistics on this statview as follows:

1. Determine the *root* table R in the set of joined tables. This table is the one that has foreign keys to other tables, but with no foreign keys to it from other tables.
2. Scan the join synopsis of R, and check the scanned tuples against the selection predicate(s) in the statview.

The join synopsis of table R can be used to collect the statistics on any statview on a set of tables whose root table is R. The algorithms presented in this paper (Section 3.2) use samples and join synopses to collect statistics.

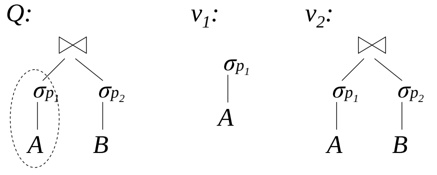

Figure 1: View benefit

1.3 Materialized vs. Statistical Views

Statistical views are similar in concept to materialized views except that materialized views contain pre-computed data, while statistical views are used only for cardinality estimation, not for query evaluation. Exploiting materialized or statistical views is based on *view matching*. The optimizer tries to match (part of) the query in question with one or more of the existing views. Matching is usually performed using the internal representation of the SQL query, which is different from one DBMS to another. If a match is found, the matched view can be used to improve the performance of that query, by reusing the result of the view (in the case of materialized views), or by improving the cost estimates, and thus getting a better execution plan (in the case of statviews).

Unfortunately, the techniques used to recommend materialized views cannot be directly adopted to recommend statviews, due to multiple fundamental differences. We mention two of these differences here:

(1) The Benefit of a View

Consider the following example: Figure 1 depicts the logical tree of a query Q, where p_1 and p_2 are some selection predicates on tables A and B, respectively. Suppose that two views v_1 and v_2 are defined. If v_1 and v_2 are materialized views, v_2 is more beneficial than v_1, since v_2 is matched with the whole query, and can be used directly to provide the results without any further processing, while v_1 matches only the circled part of Q and thus requires additional processing to obtain the results of Q. In contrast, if v_1 and v_2 are statviews, then v_1 provides an accurate cardinality estimate of the selection, allowing the optimizer to choose the appropriate join method, while v_2 provides only statistics about the top operator of Q, which generally does not help the optimizer[1]. Therefore, in this case, v_1 is more beneficial than v_2. In Section 4, we propose a benefit metric that reflects the way statviews are used.

(2) View Matching

Materialized view matching is based on *subsumption*; a materialized view is considered an exact match of a certain part of the query if the view produces the same tuples that are produced by that part of the query [23]. In case of non-exact matches (where the view produces more tuples than the query), a compensation operation is applied on the view to extract only the desired tuples. This compensation is usually in the form of applying additional predicates, joins, and/or aggregate functions to get only the desired results. The work in [23] describes the cases and conditions required for materialized view matching.

[1]Usually, input cardinalities are used for costing and planning an operation. However, some operations can be better planned knowing the output cardinality as well.

In the case of statviews, if a statview matches part of the query, the statview statistics can be used by the optimizer to accurately determine the output cardinality of this sub-query. However, the matching conditions in the case of statviews can be more relaxed than in materialized views.

EXAMPLE 1.2. Consider the following query Q and views v_1, v_2 and v_3:

```
Q:  SELECT R.e, S.c, S.f,       v1: SELECT DISTINCT R.e,
    AVG(R.d) FROM R, S, T           S.c, S.f FROM R, S, T
    WHERE R.a = S.a                 WHERE R.a = S.a
    AND S.b = T.b                   AND S.b = T.b
    GROUP BY R.e, S.c, S.f

v2: SELECT R.e, S.c,            v3: SELECT R.e, S.c, S.f
    AVG(R.d) FROM R, S, T           FROM R, S
    WHERE R.a = S.a                 WHERE R.a = S.a
    AND S.b = T.b
    GROUP BY R.e, S.c
```

Suppose that v_1, v_2 and v_3 are materialized views. v_1 cannot be matched with Q, since it does not contain the data needed to compute `AVG(R.d)`. v_2 cannot be matched with Q, since the view is only grouped by two columns, thus losing information which cannot be retrieved using any compensation. Even though v_3 matches the sub-expression involving tables R and S, it cannot be matched with Q, since the `SELECT` clause of v_3 does not include $S.b$, which is needed later on for joining the view with table T.

Now suppose that v_1, v_2 and v_3 are statviews. v_1 can provide the number of tuples produced as a result of the `GROUP BY` operation in Q, which is the same as the number of distinct combinations of the values in the three grouping columns. v_2 can give the number of groups (i.e. number of distinct values) based on the column group ($R.e, S.c$). Ideally, this information can still be useful while optimizing Q, especially if the number of distinct values in column $S.f$ is also available (from base table statistics or another statview). The concept of using partial information and assuming independence or uniformity unless otherwise known has been used in [10, 19]. v_3 can provide the exact output cardinality of joining R and S. Thus the three statviews should be considered beneficial matches during query optimization. However, note that whether these statviews are considered successful matches or not could differ from one database system to another, depending on the matching capabilities of the system, and how it utilizes statviews in query optimization. We discuss this further in Section 5.

The aforementioned differences between statviews and materialized views, as well as other special properties that we discuss in Section 2, warrant the development of a dedicated advisor that takes these special characteristics into account.

1.4 Contributions and Organization

In this paper we propose `StatAdvisor`, a system to automatically recommend statistical views that are most beneficial for a particular workload. We introduce a novel plan-based candidate enumeration technique, as well as a benefit metric that takes into account the characteristics and effect of statviews. Our work also considers the possible dependency between multiple statviews in terms of their effect on the chosen execution plan. The system amortizes the benefit of the candidate statviews across the whole workload in order to get the final recommendations. The algorithms presented in this paper have been implemented in *IBM DB2*, and used to conduct an extensive experimental study to demonstrate the effectiveness of the `StatAdvisor`, and its impact on workload performance.

The rest of this paper is organized as follows: Section 2 defines the problem of recommending statviews, and outlines the architecture of the `StatAdvisor` framework. Section 3 describes our novel plan-based candidate enumeration technique. Section 4 introduces our benefit metric, which is later used to select the final recommendations. We discuss the dependency of the `StatAdvisor` on the database engine in Section 5. Section 6 demonstrates our experimental results, Section 7 outlines related research in query optimization, and Section 8 concludes with a summary.

2. PROBLEM DEFINITION AND SYSTEM OVERVIEW

Let $W = \{Q_1, Q_2, ..., Q_n\}$ be a workload, where Q_i is an SQL query. Let c_{max} be a defined constraint (e.g. the maximum number of statviews that can be maintained in the system, or the maximum size of the materialized statistics). The problem of recommending statviews is defined as follows: *Find a set of statviews that minimizes the execution time of W while satisfying the constraint c_{max}.*

2.1 Key Insights

In this section we present a set of observations that are instrumental to our approach.

OBSERVATION 2.1. *Statviews have a direct effect on execution cost only when they help the optimizer choose a different (better) execution plan.*

A tangible benefit of statviews is the improvement in query performance. This improvement only occurs if the query optimizer chooses a different (better) execution plan based on the new statistics. Unfortunately, reducing errors in estimating the cost of (some) query predicates does not guarantee changing the current plan to a better one. Consider the case where the optimizer chooses the same plan with and without the statviews present. The estimated costs are different in the two instances, since they are computed from different sets of statistics, which might suggest that these statviews are beneficial to the query. However, effectively, since the same plan is chosen, the *actual* execution cost of the query is the same, indicating that the collected statviews are not beneficial. We consider a set of statviews beneficial only if their availability causes a plan change. Therefore, it is necessary to study the effect of specific statistics on the change in the execution plan. The use of plan change as a measure of statistics relevance has been previously used in [9] in the context of reducing a set of statistics to a necessary subset that has the same overall effect on choosing an execution plan.

OBSERVATION 2.2. *It is hard to estimate the effectiveness of statistics on workload performance without actually collecting the statistics.*

When recommending auxiliary database structures, one of the most essential tasks is to estimate the effectiveness (or benefit) of a particular structure (e.g., an index or a materialized view) on query performance. This is usually accomplished by simulating the existence of these structures, and estimating their properties using the available statistics. The cost of the query is estimated with and without the structure, and the benefit is the difference between the two cost estimates. These cost estimates are comparable, since they are both computed assuming correctness of the available statistics (which are the same in both cases), and using the same assumptions employed by the optimizer.

In the `StatAdvisor`, the structures under investigation are the statistics themselves. In contrast to indexes or materialized views, we cannot simulate the existence of statistics. If there was a method to estimate the value of a particular statistic without employing unrealistic assumptions, then such method would have been used to obtain more accurate statistics in the first place. Another possibility is estimating the extreme values of a statistic, estimating the cost of the query using both extremes, then computing the benefit as the difference between the two cost estimates (as done in [6]). However, the benefit of a statistic should not be based on the difference between query costs in extreme conditions, but rather on the difference between the plans generated with and without the statistic (Observation 2.1). We explain in Section 4 how the benefit of a statview is estimated in `StatAdvisor`.

Observation 2.3. *Statviews achieve their effectiveness in groups.*

Given a query workload, it is often the case that we cannot recommend all the beneficial statviews for all the queries (due to some constraint, as mentioned in the problem definition). Traditionally, a benefit score is assigned to each statview in isolation, and each statview might or might not be part of the final recommendations. The problem here is that, in many cases, a query benefits only if a certain set of statistics are all present, but not if one or more of them are missing. Therefore, recommending only a subset of these statviews might not introduce any benefit.

Consider the query plan at the top left corner of Figure 2. This is a plan obtained when no statviews are available. The values in parentheses represent the estimated cardinality at each operator in the plan. The circled sub-expressions represent two candidate statviews. Collecting only one of the two statviews results in changing the cardinality estimate of the corresponding sub-expression and all its parents, but it does not cause a plan change. However, collecting both statviews causes the optimizer to choose a different plan. In the extreme case, collecting a subset of the required statviews can cause the optimizer to choose an execution plan that is worse that the initial plan obtained using only base table statistics. Here the original plan may have been obtained by chance when "two wrongs made a right". Previous efforts for automated selection of statistics (e.g. [6, 9]) picked one statistic at a time, assuming independence between statistics. However, the authors recognized this as a limitation in their respective approaches.

Based on this observation, we introduce the concept of *statview-groups*. Once we identify the set of beneficial statviews for a given query, we treat these statviews as a single unit (called a statview-group). A benefit score is computed and is assigned to each statview-group as a whole, as opposed to individual statviews (more details in Section 4).

Observation 2.4. *It is expensive to collect all statviews needed to find the optimal plan.*

Before elaborating on this observation, we need to define the following terms:

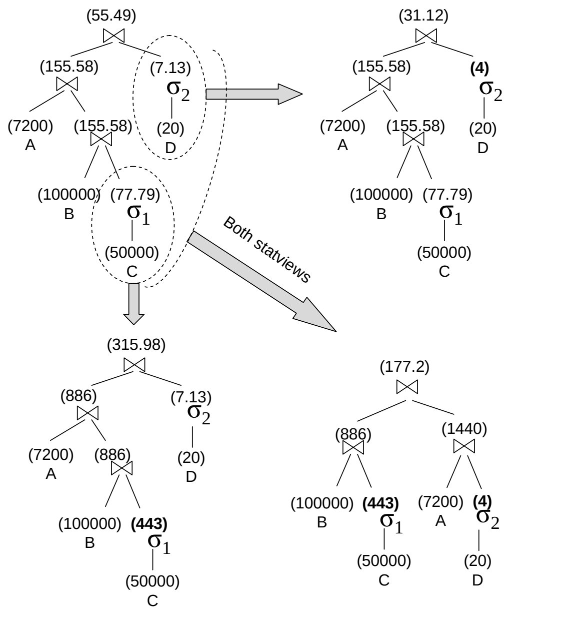

Figure 2: Effect of statview-groups

Definition 2.1. *Accurate Cost Estimate:* Consider a query plan P. The *accurate cost estimate* of P, denoted by c^P_{acc}, is the estimated cost of P if the cardinality at each operator in P is estimated without any simplifying assumptions (e.g. independence, uniformity or inclusion).

Definition 2.2. *Overestimation and Underestimation:* Let c be the estimated cost of P while employing an arbitrary number of assumptions. The cost of P is said to be *overestimated* if c is greater than c^P_{acc}. Similarly, the cost of P is said to be *underestimated* if c is less than c^P_{acc}.

For a given query, in order to guarantee getting the optimal plan, statviews that correspond to the following subexpressions are relevant and should be available:

1. Subexpressions that appear in any plan chosen by the optimizer whose cost is underestimated

2. Subexpressions that appear in any plan whose cost is overestimated, and that will be chosen by the optimizer when corrected

The first set of statviews rectifies the problem where a suboptimal plan is chosen by the optimizer because its cost is underestimated. Whenever a plan P_1 is chosen by the optimizer, collecting statistics on expressions that appear in P_1 will correct the estimated cost of this plan, but it might also cause the optimizer to choose a different plan P_2, whose cost is still underestimated and is less than the currently accurate cost of P_1. Therefore, getting the set of statviews that correct all plans with underestimated costs requires repeating this process until a stable state is reached (no new plan is chosen). The second set of statviews rectify the problem where the actual optimal plan is not chosen because its cost is overestimated, leading the optimizer to favor another plan.

Determining the first set of statviews is feasible. The plans in question are those returned by the optimizer. We only need to collect the statviews that appear in each plan

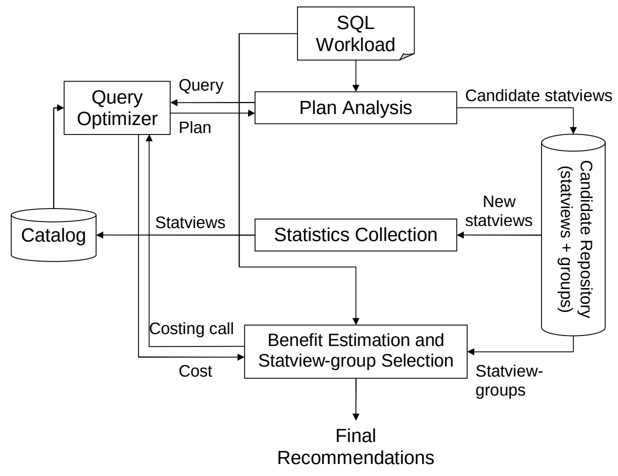

Figure 3: System architecture

returned by the optimizer, then re-optimize until no new plan is returned. This is the main idea behind our candidate enumeration technique in Section 3.1. The second set of statviews is more challenging. Plans with overestimated costs are not returned by the optimizer. Finding these plans is only possible if we search the whole plan space of the given query. This plan space can be significantly large, making it expensive to search for such plans.

However, the number of plans with overestimated costs can be reduced by examining the query structure (as opposed to specific plans), and collecting statistics on objects that exhibit certain characteristics, e.g.:

- Attributes with highly skewed distributions that appear in the query predicates
- Arithmetic expressions and user-defined functions that appear in the query predicates (e.g. the expression `Price*(1-Discount)<5000` in Example 1.1)
- Significant mismatch in the range of values in the join attributes of two relations

Candidate statviews can be generated based on these objects during an initial analysis of the query structure before the optimizer is invoked to obtain an execution plan. Recommending statviews based solely on analyzing the query structure is the technique used by current approaches [6]. Our plan-based approach is novel in the way it uses the generated plans to eliminate the possibility of choosing any plans with underestimated costs, in addition to using query analysis to reduce the space of overestimated plans.

If we only determine the first set of statviews, we guarantee that the cost of the plan chosen by the optimizer is accurate and not underestimated. This chosen plan might not be optimal, but at least there will be no surprises in its execution cost. Plans with this property are often called *predictable plans.* A related line of work is concerned with the trade-off between optimal and predictable plans [4].

Based on this observation, instead of collecting enough statviews to find the optimal plan, we opt for the more relaxed (and less expensive) objective: collecting enough statviews to guarantee getting a predictable plan. The technique we use to achieve this objective is explained in more detail in Section 3.1.

Algorithm 1 StatAdvisor(W, c_{max})

```
1: $(V, G) \leftarrow PlanAnalysis(W)$
2: while $V \neq \phi$ do
3:     CollectStatviews($V$)
4:     $(V, G) \leftarrow PlanAnalysis(W)$
5: end while
6: EstimateBenefit($G$)
7: $R \leftarrow$ StatviewGroupSelection($G, c_{max}$)
8: Return($R$)
```

2.2 StatAdvisor Framework

We adopt a benefit-cost analysis to recommend the most beneficial statviews (subject to the constraint c_{max}). This approach is similar in concept to most database design advisors [3, 22, 24]. However, the computation of the benefit estimates takes into account the special characteristics of statviews and their effect on query performance.

Figure 3 depicts the general framework of the `StatAdvisor`. The system takes as input a workload of SQL queries, and outputs a set of recommended statviews. The *Plan Analysis* module is responsible for finding the candidate beneficial statviews for each query in the workload. This is achieved by invoking the optimizer for each query in the workload and analyzing the returned execution plan (Section 3). The candidate statviews are stored in the *Candidate Repository*, grouped into statview-groups (based on which queries generated them). The *Statistics Collection* module takes a list of candidate statviews, creates and collects statistics on these statviews, and stores the collected statistics in the system catalog, to be used by the query optimizer. The *Benefit Estimation and Statview-group Selection* module assigns a benefit score to each statview-group, based on the plan change in its corresponding query, then chooses a subset of the candidate statview-groups that maximizes the benefit while satisfying the predefined constraint (Section 4).

Algorithm 1 gives a high-level overview of our approach. The algorithm starts by analyzing the workload W, and obtaining the set of candidate statviews V, partitioned into a set of statview-groups G (line 1). Subsequently, the algorithm performs a number of iterations while V is not empty (lines 2-5). In each iteration, the statistics on the statviews in V are collected and added to the catalog, then the plan analysis module is re-invoked. Finally, a benefit score is assigned to each statview-group, and the algorithm selects the groups with the maximum benefit that satisfy the constraint c_{max}, compiling them into the recommendation set R.

3. PLAN-BASED CANDIDATE ENUMERATION

In this section, we discuss our approach for finding the candidate statviews that can cause a plan change in the given queries. First we describe the technique for a single query in Section 3.1, then we extend it to process the whole workload in Section 3.2.

3.1 Candidate Enumeration for a Query

Given a query Q, let P_0 denote the plan chosen by the optimizer in the absence of any statviews. Due to cardinality estimation errors, the estimated cost of P_0 is likely to be inaccurate, which means that P_0 might not be a good choice for executing Q. As mentioned in Observation 2.4, our objective is to ensure that the cost of the chosen plan

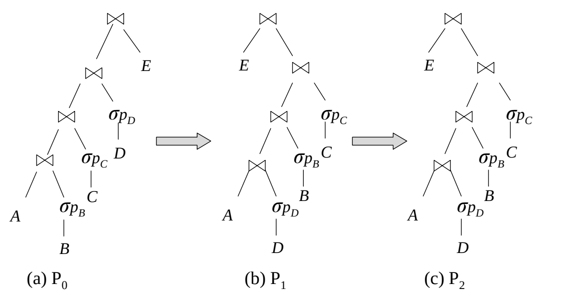

Figure 4: Candidate enumeration

is accurately estimated, in which case the chosen plan is predictable. To achieve this, we need to have accurate cardinality estimates for all the sub-expressions that appear in P_0. Each of these sub-expressions corresponds to a statview. Collecting all the statviews that appear in P_0 gives a better estimation of the cost of P_0. In most cases, collecting all statviews is unnecessary and extremely expensive. Hence, we define *important statviews* as follows:

Definition 3.1. Given a query plan P, the *important statviews* in P are statviews that correspond to logical expressions in P that involve one or two tables, along with the corresponding selection predicates.

In our experiments (Section 6.2), we found that collecting only the important statviews provides enough accuracy, and collecting any more statviews increases the running time of the `StatAdvisor` without introducing significant benefit.

After collecting the important statviews in P_0, we re-optimize the query. There are two possible scenarios: the optimizer might choose the same plan, or it might choose a different one due to changes in cost estimates. If the optimizer chooses the same plan, the collected statviews were not necessary, since they caused no plan change (cf. Observation 2.1). On the other hand, if the optimizer chooses a different plan, this might be due to underestimating the cost of the new plan, which penalizes the old plan for having a more accurate cost estimate. Therefore, we need to repeat the process for the new plan P_1. The process is repeated until either: (1) we reach a plan that has been encountered before (since all the important statviews of that plan would have already been collected, and its cost would be accurately estimated), or (2) the important statviews of the current plan have already been collected. For example, consider the plan P_0 in Figure 4(a). After collecting the important statviews in P_0 and re-optimizing, the obtained plan is P_1 (Figure 4(b)). After another iteration, the plan obtained (P_2) is the same as P_1, thus the algorithm terminates.

The set of all statviews collected so far are the candidate statviews for the query, since they are necessary to reach the final plan. As mentioned in Observation 2.3, all the candidate statviews for the query are treated as a single unit (statview-group) from now on, since they are all needed to achieve the objective.

This technique guarantees that all plans that have been chosen at some point are accurately estimated, and the last chosen plan P is indeed the best one among them. Let P' denote a plan that has never been chosen by the optimizer. P' could be any of the following:

1. The cost of P' is accurately estimated, thus it is indeed worse than P (P is favored by the optimizer over P')
2. The cost of P' is underestimated, thus it is still worse than P (since the estimated cost of P' is already greater than that of P)
3. The cost of P' is overestimated, which means it could possibly be better than P. However, as discussed in Observation 2.4, finding P' requires searching the whole plans space.

Using this technique, we will miss getting the optimal plan if its cost has been overestimated by the optimizer, due to overestimating the cardinality of a certain sub-expression whose cardinality has not been collected (cf. Observation 2.4).

3.2 Candidate Enumeration for a Workload

The naïve approach to obtain the candidates for the whole workload is to repeat the technique described in Section 3.1 for each query. The problem with this simple approach is that the technique includes actual collection of statistics, and repeating it for each query will result in accessing the same data pages multiple times. This can become a performance issue because of the repeated I/O operations on the same disk pages. To overcome this problem, we process all the queries in the workload in parallel, and employ *batch statview collection*; statviews that involve the same set of tables are collected simultaneously from the same table sample or join synopsis [2], thus causing the respective sample or join synopsis to be read from disk only once. Algorithm 2 gives our approach to produce the candidates with respect to the whole workload W of n queries. Algorithm 2 provides the details of lines 1-5 in Algorithm 1. For each query Q_i, the algorithm returns V_i, a statview-group containing the candidate statviews for Q_i.

The algorithm starts with an initialization step (lines 2-9). For each query Q_i, a flag $finished_i$ is set to false (indicating that processing Q_i is not finished), Q_i is optimized producing plan $P_{i,0}$, and the important statviews in that plan are determined (VT_i). The iterative part of the algorithm is in lines 10-37. In each iteration, the set V is the union of all VT_i for the queries that are still being processed (i.e. V is the set of all newly discovered statviews). V is partitioned into disjoint sets $U_1, ..., U_m$, such that the statviews in U_j are all on the same table(s) but most likely with different selection predicates. For each set U_j, a sample or a join synopsis is collected (depending on how many tables are involved) and all statviews in U_j are collected simultaneously. Note that if the required sample already exists (from previous iterations), it is not recreated. The tuples in the sample are scanned only once, and checked against the selection predicates in each statview. If a tuple satisfies the selection predicate of statview $v \in U_j$, this tuple takes part in computing the statistics on v. After all the statviews are collected, the queries are re-optimized, and the obtained plans are examined. If a query produces the same plan, or a different plan whose important statviews have already been collected, processing is stopped for that query (as in Section 3.1). Only the remaining queries are entered into the next iteration. Processing stops when no queries are left.

Ideally, we would know beforehand which statviews we need to collect on each query in the workload. In that case, we would group all statviews that have the same set of tables, and collect them simultaneously from the corresponding sample, thus only touching that sample once. Our experiments show that most queries require 1-3 iterations before

Algorithm 2 CandidateEnumeration(W)

```
1: Assume that $W = \{Q_1, Q_2, ...Q_n\}$
2: for $i = 1$ to $n$ do
3:     $finished_i \leftarrow false$
4:     Optimize $Q_i$, let $P_{i,0}$ be the chosen plan
5:     $V_i \leftarrow \{\}$
6:     $VT_i \leftarrow FindImportantStatviews(P_{i,0})$
7: end for
8: $k \leftarrow 1$
9: $done \leftarrow false$
10: while $\neg done$ do
11:     $V = \{\bigcup_i VT_i | finished_i = false\}$
12:     Partition $V$ into $m$ disjoint sets s.t. each set $U_j$ contains statviews on the same set of tables
13:     for $j = 1$ to $m$ do
14:         if $U_j$ is on a single table then
15:             Get a sample of that table
16:         else{$U_j$ is on multiple tables}
17:             Get a join synopsis
18:         end if
19:         Collect all statviews in $U_j$
20:     end for
21:     $done \leftarrow true$
22:     for $i = 1$ to $n \wedge finished_i = false$ do
23:         $done \leftarrow false$
24:         Optimize $Q_i$, let $P_{i,k}$ be the chosen plan
25:         if $P_{i,k} = P_{i,l}$ for some $l < k$ then
26:             $finished_i \leftarrow true$
27:         else(different plan)
28:             $V_i \leftarrow V_i \cup VT_i$
29:             $VT_i \leftarrow FindImportantStatviews(P_{i,k})$
30:             $VT_i \leftarrow VT_i - V_i$
31:             if $VT_i$ is empty then
32:                 $finished_i \leftarrow true$
33:             end if
34:         end if
35:     end for
36:     $k \leftarrow k + 1$
37: end while
```

termination. The convergence of the algorithm is addressed in detail in our experimental evaluation (Section 6.2). We observed that the running time of the algorithm grows linearly with the number of queries in the workload as well as with the number of tables referenced in these queries.

For a given query, our candidate enumeration algorithm finds the set of statviews that are guaranteed to make the optimizer choose an execution plan with an accurately estimated cost. However, the statviews in the produced statview-group might be more than what is actually needed to get the final predictable plan. As a result, it is possible to reduce the statview-group to the minimal set of *necessary* statviews that give the same result. This can be accomplished using a similar concept to the *shrinking set* algorithm presented in [9]; which takes as input a set of statistics, and gives a subset of this set that has the same overall effect. The algorithm starts with the whole set, and checks whether the removal of any statview from the set would change the produced plan. A statview whose removal does not affect the produced plan is not necessary and can be safely discarded. The algorithm repeats until no more statviews can be removed.

4. BENEFIT ESTIMATION AND STATVIEW SELECTION

As mentioned in Section 2, the objective is to choose the statviews that have the maximum benefit (minimize the workload execution time) while satisfying the cost constraint. As a result, we need to define a benefit metric for statview-groups that captures the saving in the execution time. In Section 4.1, we define the benefit of a statview-group to a particular query, and in Section 4.2, we extend that definition to the whole workload. Section 4.3 presents our selection algorithm that exploits these benefit estimates.

4.1 Benefit for a Single Query

The most accurate and intuitive metric for measuring the benefit of a statview-group V to a query Q is the reduction in the execution cost of Q as a result of using V in the optimization. Let P_0 be the plan chosen by the optimizer when no statviews are present, and let P_V be the plan chosen when V exists. The benefit $B(V, Q)$ can be expressed as:

$$B(V, Q) = ActCost(Q, P_0) - ActCost(Q, P_V) \quad (1)$$

where $ActCost(Q, P)$ is the *actual* execution cost of Q using plan P. $B(V, Q)$ represents the saving in the execution cost. Note that the difference in costs is primarily due to the change of execution plans triggered by the presence of more accurate statistics (statviews) in V. If the statistics provided by V are not significant enough to cause a plan change, then $P_0 = P_V$, and $B(V, Q) = 0$. Computing $B(V, Q)$ requires compiling and executing Q twice (with and without V present), to get the actual execution cost in each case. This is infeasible for a large workload with complex queries.

To avoid having to execute the query twice, a possible approach is to use the *estimated* cost of the query instead of the *actual* cost. This is based on the assumption that the estimated cost is monotonic in the actual execution cost. For a given query plan P and a statview-group V, let $EstCost(P, V)$ denote the estimated cost of P in the presence of V. The monotonicity assumption between the estimated and actual execution cost implies that given a query Q, two plans P_1 and P_2, and a statview-group V, if $EstCost(P_1, V) > EstCost(P_2, V)$, then $ActCost(Q, P_1) > ActCost(Q, P_2)$. However, benefit estimation using the estimated costs is not straightforward. In other words, we cannot use the difference in estimated cost with and without the statviews. This is because the estimated cost without the statviews is computed based on inaccurate statistics, hence it is not comparable to the estimated cost with the statviews (as they are computed using different sets of statistics). To illustrate this problem, consider the following example.

EXAMPLE 4.1. The optimizer is invoked without statviews and the output is plan P_0 with estimated cost $c_0 = 100$. The optimizer is invoked once again, with the statviews present, and it produces plan P_V with estimated cost $c_V = 150$. At first glance, this might indicate that the presence of the statviews harmed the query. However, in reality, it is important to examine the chosen plans themselves, and not just the estimated execution cost. In the second invocation of the optimizer (with V present), the estimated cost of P_V is clearly lower than that of P_0 (since P_V was favored by the optimizer over P_0). Therefore, we can only compare the plan costs estimated with the same set of statistics.

Based on this observation, we can compute an approximate benefit as follows:

$$B'(V, Q) = EstCost(P_0, V) - EstCost(P_V, V) \quad (2)$$

Computing B' requires only optimizing Q twice; the first time to obtain P_0, and the second time to obtain P_V as well as the cost of both plans. This eliminates the need to execute Q while providing an approximate benefit score for V. Note that, in our solution, the plans and their respective costs are already obtained as part of the candidate enumeration process, so we need only one additional call to the optimizer's costing functions to get $EstCost(P_0, V)$. Again, if the optimizer chooses the same plan both times, then $P_0 = P_V$, and $B'(V, Q) = 0$.

4.2 Benefit for a Workload

Given the benefit of statview-groups to individual queries, it is easy to compute the benefit of these statview-groups to the whole workload. Consider a statview-group V. Let $W_V \subset W$ be the set of queries that generated V (i.e. V has been separately generated by each query in W_V). The benefit of V to the workload W can be computed by summing the benefits of V to each query in W_V, or more formally:

$$B(V, W) = \sum_{Q \in W_V} B'(V, Q) \tag{3}$$

Note that the benefit of V for each individual query is independent from the other queries, therefore they can be safely added. From this point on, we shall write $B(V, W)$ simply as $B(V)$.

4.3 Statview-Group Selection

At this point, we have a set of statview-groups $G = \{V_1, ..., V_n\}$ where n is equal to the number of queries in the workload. For a given statview-group V_i, $B(V_i)$ and $C(V_i, R)$ denote the benefit and cost of V_i respectively. The cost can be computed differently depending on the database system and the computing environment. For example:

- In systems where the main concern is storage space, the cost can represent the space needed to store the statview and its statistics. Since the statistics are actually collected as part of the enumeration phase, it is not hard to determine how much space they occupy.
- If the main concern is query optimization speed, then the fewer statviews maintained by the system, the better (since fewer statviews are considered for matching). In this case, the cost of all statviews is the same (can be set to 1). Note that even if all statviews have the same cost, the cost of statview-groups is different, since the number of statviews in each group is arbitrary.

Note that $C(V_i, R)$ is also a function of the recommendation list R, since the statview-group V_i might contain some statviews that have already been added to R, and do not introduce any extra cost. For any two statview-groups V_i, V_j:

- $Benefit(V_i, V_j) = B(V_i) + B(V_j)$, since the two statview-groups benefit different queries
- $Cost(V_i, V_j) \leq C(V_i, R) + C(V_j, R)$, since there might be some common statviews, and their cost should not be counted more than once

This problem is a generalization of the *0/1 knapsack problem*, where the cost of an item is not constant, but depends on the items chosen. The exact solution is exponential. However, we can use the same greedy algorithms used for knapsack, but taking care to dynamically modify items' costs based on the items chosen so far.

Algorithm 3 StatviewGroupSelection(G, c_{max})

```
1: $c \leftarrow 0, R \leftarrow \phi$
2: while ($|G| > 0 \wedge c < c_{max}$) do
3:     $V_{best} \leftarrow null$
4:     $B_{best} \leftarrow 0$
5:     for all $V \in G$ do
6:         if $B(V) > B_{best} \wedge C(V, R) \leq c_{max} - c$ then
7:             $V_{best} \leftarrow V$
8:             $B_{best} \leftarrow B(V)$
9:         end if
10:    end for
11:    if $V_{best} \neq null$ then
12:        $c \leftarrow c + C(V_{best}, R)$
13:        $G \leftarrow G - V_{best}$
14:        $R \leftarrow R \cup V_{best}$
15:    else
16:        Break
17:    end if
18: end while
19: Return($R$)
```

In our implementation, we use the greedy solution given in Algorithm 3. The algorithm takes as input the set of candidate statview-groups $G = \{V_1, ..., V_n\}$, as well as the user-defined constraint on the maximum cost (c_{max}). The output of this module is the set R of final recommendations, where $R \subseteq V_1 \cup ... \cup V_n$.

The algorithm is iterative. At each iteration, the algorithm tries to find the statview-group with the maximum benefit that can still fit within the constraint. If such statview-group is found, its contents are added to the recommendation list R. The algorithm has a polynomial running time in the number of candidate statview-groups (which is almost equal to the number of queries in the workload).

5. DEPENDENCY ON DATABASE ENGINE

The implementation details and some algorithms of `StatAdvisor` depend mainly on the underlying database engine. This is because currently the way statviews are defined and utilized is not standard across DBMSs. Specifically, the engine-dependent module of `StatAdvisor` is the plan analysis module. All the remaining modules are independent of the database engine and how statviews are used.

The plan analysis module is mainly affected by the statview matching and utilization capabilities of the database engine. In the extreme case, if the database engine does not support statview matching, then the execution plans will never change no matter what statviews are created, and the plan analysis module will not produce any candidates. For an engine that supports statview matching, the plan analysis module must be aware of how the matching is performed in order to come up with the candidate statviews. Some of the criteria that have to be considered include:

- Whether or not the matching has to be exact (the statview has to be strictly equivalent to the sub-expression being matched)
- For non-exact matching, what differences can exist while still resulting in a successful match (e.g., partial set of predicates, different output columns)
- Whether or not multiple partial selectivities can be

obtained from several statviews to estimate the selectivity of one sub-expression

- What statistics on the statviews can be used by the optimizer (e.g., only the number of tuples in the statview, histograms on statview columns, etc.)

Producing candidates that are not matchable means that the optimizer will not have access to any usable new statistics, and will choose the same plan.

6. EXPERIMENTAL EVALUATION

In this section, we present experimental results of an implementation of the `StatAdvisor` in *IBM DB2*.

6.1 Setup

Data: We carried out our experiments on two different data sets. The first data set, DS_1, is a version of the TPC-DS [21] data set with scale factor 1. The second data set, DS_2, is a synthetic database with six relations `CAR, OWNER, DEMOGRAPHICS, ACCIDENTS, LOCATION,` and `TIME`. The size of this data set is 1 GB. Several primary-key-to-foreign-key relationships exist between the tables. Each table is composed of four to eight attributes. Some attributes are uniformly distributed and others are more skewed. A number of correlations between attributes, such as `Make` and `Model`, are inherent in the attribute definitions.

Workloads: We used two workloads for our experiments. The first workload, W_1, consists of 23 queries from the TPC-DS benchmark (queries 3, 7, 9, 12, 13, 15, 19, 20, 26, 42, 43, 44, 48, 52, 55, 62, 75, 76, 82, 84, 91, 98 and 99). These queries were selected because they do not contain subqueries[2]. However, the selected queries include various constructs, e.g. arithmetic expressions, functions, equi-joins, range joins, equality and range filtering predicates, conjuncts and disjuncts. Each query consists of three to seven joined tables. The second workload, W_2, corresponds to the second data set (DS_2), and contains 100 synthetically generated SPJG queries. Each query consists of one to five joined tables, and several selection predicates, some of which are correlated. Some of the queries also include aggregate functions and grouping. An example query is:

```
SELECT city, COUNT(*) FROM owner o, car c
WHERE c.ownerid = o.id
AND c.make = 'Honda' AND c.model = 'Civic'
GROUP BY city
```

6.2 Candidate Enumeration

As mentioned in Section 3.1, given a particular query plan, the plan analysis module determines the *important* statviews for this plan. Assume that we let the `StatAdvisor` collect all statviews in the given plan that include up to t tables. Figure 5(a) depicts the running times of the `StatAdvisor` for different values of t, when it is invoked for workload W_1, as well as the running times of W_1 given the obtained recommendations for each value of t. The `StatAdvisor` running time increases almost linearly with t. On the other hand, the major performance improvement for

[2]At the time of writing this paper, *DB2* does not match statviews with queries that include subqueries. However, `StatAdvisor` can generally recommend statviews that correspond on any query expression, including subqueries

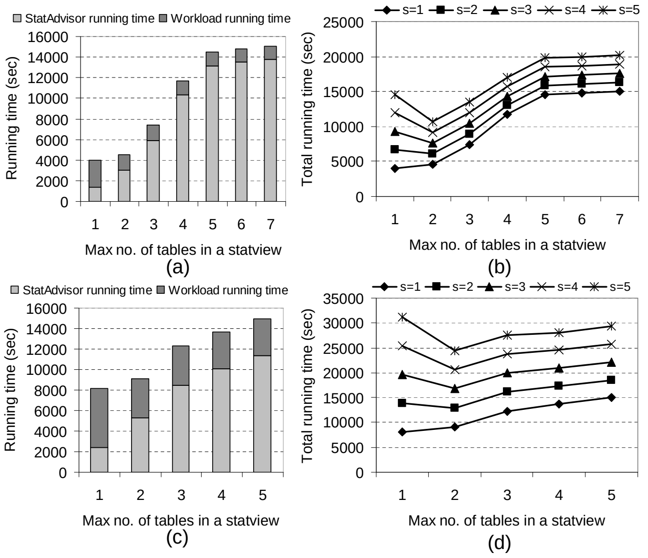

Figure 5: Important statviews

the workload occurs at $t = 2$, i.e. when the system collects statviews with up to 2 joined tables. As t increases further, there is slight improvement, but it is overweighed by the increase in the running time of the `StatAdvisor`. If the workload is executed more than once, it becomes even more evident that the best performance is achieved at $t = 2$. Generally, the total running time = (`StatAdvisor` running time + s * workload execution time), where s is the number of executions of the workload. Figure 5(b) depicts the total running time against t, for different values of s. The best performance for most values of s occurs at $t = 2$. Figures 5(c) and 5(d) depict the corresponding results for the second workload W_2, in which the same effect can be seen. Based on these results, we decide to limit the important statviews to those with one or two joined tables, along with their selection predicates, since including more statviews increases the overhead without introducing significant benefit.

Figure 6 shows the convergence of the candidate enumeration algorithm when (1) all statviews in a plan are collected, and (2) only the important statviews in a plan (cf. Definition 3.1) are collected. The x-axis represents the various iterations, and the y-axis represents the number of queries being processed in each iteration. Figures 6(a) and 6(b) correspond to workloads W_1 and W_2, respectively. When all statviews are collected, the algorithm terminates after 4 iterations, whereas it terminates after 2 or 3 iterations when only the important statviews are collected. The reason behind this behavior can be explained as follows:

Consider a query Q. At the k^{th} iteration, the algorithm is processing the plan P_k. Assume that we collect only the important statviews, and that re-optimizing the query still results in the same plan P_k. Let $C_{imp}(P_k)$ be the estimated cost of P_k using the important statviews. In an alternative scenario, assume that we collect all statviews in P_k. Let $C_{all}(P_k)$ be the estimated cost of P_k in this case. Since the important statviews capture most of the estimation error, the estimated cost should not change drastically by collecting all statviews. Therefore $C_{imp}(P_k)$ and $C_{all}(P_k)$ are usually very close. However, even though they are close, $C_{all}(P_k)$ can be slightly higher than $C_{imp}(P_k)$. In this case,

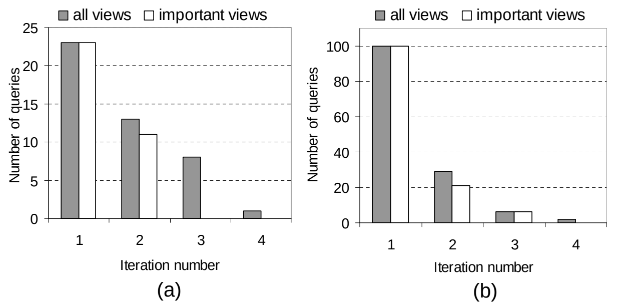

Figure 6: Convergence of candidate enumeration

another plan P_{k+1} whose estimated cost $C(P_{k+1}) < C_{all}(P_k)$, will be chosen by the optimizer, leading to more iterations.

Note that since P_{k+1} was not chosen in the first scenario, this means that $C_{imp}(P_k) < C(P_{k+1})$. In other words $C_{imp}(P_k) < C(P_{k+1}) < C_{all}(P_k)$. And since $C_{imp}(P_k)$ and $C_{all}(P_k)$ are very close, the estimated cost of P_{k+1} cannot be significantly less than that of P_k. Therefore, even though a different plan is chosen, the performance gain is negligible.

6.3 Overall Workload Performance

The test workloads are executed in the following settings:

1. Statistics are available on all base tables and their attributes, including table cardinalities, number of distinct values in each column, etc. This represents the common case in most database systems.

2. Statistics are available on base tables and their attributes, plus statistics on the statviews recommended by the `StatAdvisor` for this particular workload.

The `StatAdvisor` recommended 35 statviews for workload W_1, and 50 statviews for W_2 (given that no user limitations were set). Generally, the recommended statviews can be categorized into two main types:

Statviews on single tables: These eliminate the errors in single-table expressions. Such errors usually arise from correlation between predicates on the same table.

Statviews on two joined tables: When local predicates are applied to one or two tables, the distribution of the results is usually very skewed. It is hard to estimate the size of joining these two result sets, and the estimation error is usually very large. Thus, having statviews on joined pairs of tables eliminates these errors.

An example of a query that benefited from the recommended statviews is the second query in W_1 (TPC-DS query 7). The query includes 3 filtering predicates on the `Customer_Demographics` table, 1 predicate on the `Date_Dim` table, and 2 disjunctive predicates on the `Promotion` table. Figure 7(a) shows the plan chosen for this query with no statviews in the system. The values in parentheses represent the estimated cardinality at each operator in the plan. The recommended statviews based on this query are:

v_1: $\sigma_1(Date\_Dim)$

v_2: $\sigma_2(Promotion)$

v_3: $\sigma_3(Customer\_Demographics)$

v_4: $Store\_Sales \bowtie (\sigma_1(Date\_Dim))$

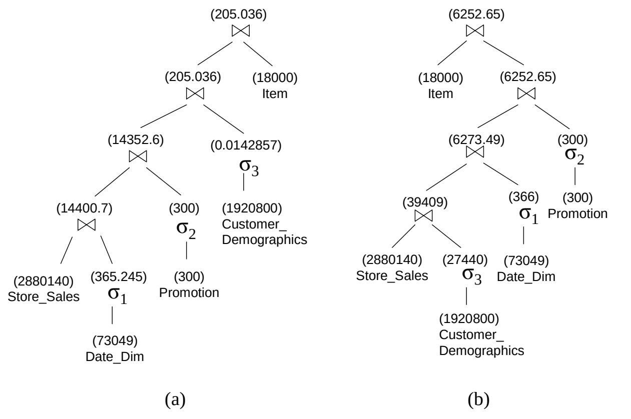

Figure 7: Execution plans for TPC-DS query 7

Figure 7(b) shows the plan chosen for this query after creating the statviews. Note the large estimation error in the expression corresponding to v_3 (due to the correlation between the 3 predicates on `Customer_Demographics`). Also, the expression corresponding to v_4 had a large estimation error resulting from applying the selection predicate on `Date_Dim`), which results in having skewed data, then joining this data with `Store_Sales`). However, the expression corresponding to v_4 does not appear in the final plan (the actual cardinality of this expression is 553,476). Collecting v_4 provides an accurate estimate for this expression, improving the query execution time from 610 to 106 seconds (5.5 times faster).

Figure 8(a) is a box plot (a graph depicting the smallest observation, lower quartile, median, upper quartile and largest observation) of the execution time of the queries of W_1 in the two settings. Figure 8(b) shows a scatter plot of the elapsed times of the individual queries of W_1, where the x-axis represents the time in the first setting, and the y-axis represents the time in the second setting. Some queries lie in the degradation region, since the presence of base table statistics is often sufficient to get accurate estimates, and the presence of statviews only introduces extra overhead. However, for most queries, the overhead introduced by the statviews is outweighed by the gain in performance as a result of better statistics, and hence, a better execution plan. Figures 8(c) and 8(d) represent the corresponding results for W_2. As can be seen, the introduction of statviews significantly improves the performance of the system.

We conducted this same experiment with indexes available for both workloads. The indexes were obtained by running the *DB2 Design Advisor* [22, 24] on each workload and materializing the recommended indexes. We then ran each workload with and without the recommended statviews obtained from `StatAdvisor`. The performance improvement was similar to the no-indexes scenario, and is not shown here for lack of space.

6.4 Comparison with Previous Work

For this experiment, we implemented the statistics selection algorithm given in [6]. We shall refer to this implementation as `SITadvisor`. We invoked both our `StatAdvisor` and `SITadvisor` for the workload W_2, with no limitations on the number or size of the recommended statviews. We then executed the workload given each set of recommendations and recorded the execution time for each query in both cases. We were unable to test `SITadvisor` with the workload W_1 because the queries in W_1 involve constructs

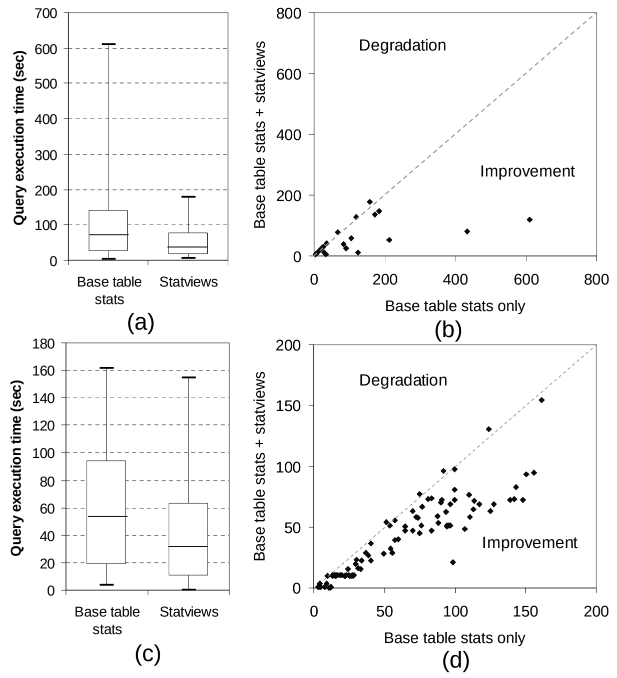

Figure 8: Workload performance

like arithmetic expressions, range joins, and disjuncts, while `SITadvisor` only supports queries with conjuncts of predicates, equi-joins, and selection predicates on table columns (not on arbitrary arithmetic expressions).

Table 1 gives a summary of the differences between `SITadvisor` and `StatAdvisor`. Note that `SITadvisor` does not collect statistics, so we added the collection time of its recommended statistics to make the comparison with `StatAdvisor` possible. `StatAdvisor` takes longer to run, since it collects more statistics than it needs to determine whether or not there is a plan change. However, it produces almost 50% fewer statviews than `SITadvisor`. The workload performs 32% better given the `StatAdvisor` recommendations than it does given the `SITadvisor` recommendations.

Figure 9 shows the individual execution times of each query in the workload in the two cases. The x-axis is the execution time given the `SITadvisor` recommendations while the y-axis is the execution time given the `StatAdvisor` recommendations. Most queries run faster with the `StatAdvisor` recommendations, since `SITadvisor` assumes predicate independence and does not recognize statview-groups.

7. RELATED WORK

A considerable amount of research has addressed the impact of database statistics on query performance. The approaches that tackle the statistics aspect of cost-based optimization can be categorized mainly as being either *reactive* or *proactive*. Reactive approaches are based on monitoring a query *during* execution, and reacting to observed errors between the initial estimates and the actual values from the query feedback. One possible approach is to use the error as an adjustment factor to correct statistics for future queries [20], or to trigger statistics collection if it exceeds a certain threshold [1]. A different approach is to react to errors by re-optimizing the current query [14, 17]. In contrast, proactive approaches try to predict, identify and possibly solve potential problems by doing additional work

Table 1: `SITadvisor` vs. `StatAdvisor`

	`SITadvisor`	`StatAdvisor`
Advising + Collection time	4045 sec	5309 sec
No. of statviews	102	50
Workload running time	5701 sec	3876 sec

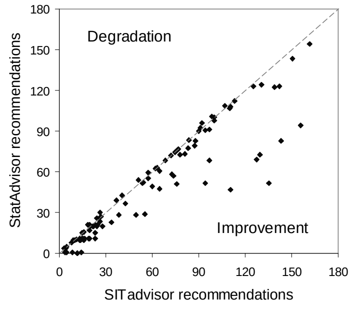

Figure 9: `SITadvisor` vs. `StatAdvisor`

before query execution. Babu et al. [5] proposed a system that enumerates and maintains different plans, then chooses among them during execution based on query feedback. A different approach [9] performs sensitivity analysis in order to decide which statistics to collect so that the optimizer will have enough information to optimize a given query, and collects these statistics during query optimization. The work in [10] introduced a system that also collects statistics on the fly. However, the statistics collected are Query-Specific Statistics (QSS) as opposed to general statistics.

A related line of work is that concerned with constructing views for statistics collection purposes. Bruno and Chaudhuri [6, 7] extend traditional optimizers to exploit statistics built on expressions corresponding to intermediate nodes of query plans. View matching techniques are used to decide which views to use. The use of views provide more accurate statistics and reduces the need to employ simplifying assumptions. Larson et al. [15] proposed using *Sample Views* for cardinality estimation. Sample views are similar in concept to statviews except that they contain a sample of the actual data, and its statistics. The work in [15] is mostly concerned with maintaining the sample views to keep them up-to-date with data activity. Resampling is triggered when query feedback shows that a certain view has become stale.

Building and maintaining query performance enhancers, such as database indexes and materialized views has been extensively studied. The *Design Advisor* in *IBM DB2* [22, 24] analyzes a given workload, and builds a list of candidate views and indexes for each query using some heuristic rules, and simulates the existence of these indexes and views. The indexes/views that are actually used in the plan given by the optimizer are recommended for this particular query. A benefit/cost analysis is used to select from these candidates. A slightly different approach is used in *Microsoft*® *SQL Server* [3]. First the system obtains the candidates in a similar manner. Subsequently, the system attempts to generate more candidate views by merging similar views that are recommended for different queries, and a benefit/cost analysis is used to determine the final workload-level recommendations. Generating views that can be used by multiple queries has also been an ongoing research problem [16].

The closest work to our proposal is the work in [6], which includes automated recommendation of SITs (statistics on intermediate tables). First, filtering predicates are investigated one at a time. For the predicate under investigation, the system uses "min" and "max" estimation strategies to estimate the extreme values of the predicate's selectivity, and propagate these values throughout the query join graph. The optimizer is invoked twice for the query on hand (for each predicate); once given all the minimum selectivity estimates, and another time given all the maximum selectivity estimates. The estimated cost in each case is recorded. If the difference between the two estimated costs exceeds a certain threshold, then the predicate under investigation is considered a candidate. The algorithm proceeds to find the generating queries for the SITs. For each candidate predicate, the algorithm examines the propagation of uncertainty in the selectivity estimation through the joins in the query graph. The join that contributes the most to the uncertainty in the whole query is the one used as the SIT for this particular predicate. A benefit score is assigned to each candidate SIT, and the system selects the SITs with the highest scores that fit within a predefined constraint. This approach does not recognize plan change as the main cause of performance gain as in Observation 2.1. In addition, it has no way of finding which SITs should be grouped together in order to cause such performance gain (Observation 2.3).

Other approaches tackling different aspects of the query optimization problem include generating robust plans that are resilient to estimation errors [4], or detecting the correlation in the data to avoid the independence assumption [12].

8. CONCLUSION AND FUTURE WORK

Collecting statistics on statviews gives the optimizer more accurate cardinality estimates at various points in the query plan, thus allowing the optimizer to estimates more accurate costs and consequently make better decisions in choosing execution plans. Choosing the correct statviews is crucial, since having too many statviews increases optimization time and administration overhead to collect statistics. `StatAdvisor` recommends the most beneficial statviews by analyzing the workload and using a plan-based candidate enumeration approach, along with a benefit/cost analysis to prune the candidates and choose the final recommendations.

For future work, we are interested in studying view matching techniques in more detail, and investigating the differences between materialized view matching and statview matching, ultimately devising specialized techniques that take into consideration the special characteristics of statviews.

9. REFERENCES

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

[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *SIGMOD*, 1999.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.

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

[5] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-optimization. In *SIGMOD*, 2005.

[6] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *SIGMOD*, 2002.

[7] N. Bruno and S. Chaudhuri. Conditional Selectivity for Statistics on Query Expressions. In *SIGMOD*, 2004.

[8] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *SIGMOD*, 1999.

[9] S. Chaudhuri and V. Narasayya. Automating Statistics Management for Query Optimizers. *IEEE TKDE*, 13(1), 2001.

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

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

[12] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *SIGMOD*, 2004.

[13] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, 1991.

[14] N. Kabra and D. J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *SIGMOD*, 1998.

[15] P.-Å. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality Estimation Using Sample Views with Quality Assurance. In *SIGMOD*, 2007.

[16] W. Lehner, B. Cochrane, H. Pirahesh, and M. Zaharioudakis. fAST Refresh using Mass Query Optimization. In *ICDE*, 2001.

[17] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing through Progressive Optimization. In *SIGMOD*, 2004.

[18] F. Olken and D. Rotem. Simple Random Sampling from Relational Databases. In *VLDB*, 1986.

[19] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. In *ICDE*, 2006.

[20] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, 2001.

[21] TPC. TPC-DS Benchmark, *http://www.tpc.org/tpcds*, 2005.

[22] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, 2000.

[23] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering Complex SQL Queries using Automatic Summary Tables. In *SIGMOD*, 2000.

[24] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004.

Trademarks