Query Optimization in Dynamic Environments

by

Amr El-Helw

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

© Amr El-Helw 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Most modern applications deal with very large amounts of data. Having to deal with such huge amounts of data is in itself a challenge. This challenge is complicated even more by the fact that, in many cases, this data is constantly changing and evolving. For instance, relational databases that handle the data of day-to-day transactional applications often have tables with very high data change rates. It is not uncommon to even have temporary or volatile tables that get created from scratch and completely dropped over the course of one query workload.

This dissertation focuses on optimizing structured queries over dynamic and constantly changing data sets. Our work address this issue, and some of the challenges related to it.

We address the issue of database statistics becoming stale and inaccurate due to constantly changing data. We introduce ways to automatically analyze the existing statistics and recommend and collect the necessary statistics to optimize a single query or a query workload.

We introduce a mechanism to automate the recommendation and collection of statistical views for a given query workload. We also compare two methods of using these statistical views in selectivity estimation. We evaluate our methods and techniques with experimental studies using prototypes that we built into commercial database systems.

Acknowledgements

My sincerest gratitude goes to my advisor, Prof. Ihab Ilyas for his guidance, support, and encouragement over the course of my doctoral studies. Prof. Ilyas kept on challenging me, encouraging me, and pushing me to go above and beyond every challenge that I have faced, overall teaching me to become a better researcher.

I am extremely thankful to my thesis committee members, Prof. Tamer Ozsu, Prof. Ken Salem, Dr. Lukasz Golab, and Dr. Glenn Paulley. Being able to discuss my research with such an outstanding group of researchers is a great honour.

I would also like to express my gratitude to Calisto Zuzarte for his contributions and help with my research. The discussions I had with him were always very insightful and helped guide me to the right track.

Last but not least, I would like to thank my parents for their ongoing and endless love and support, without which this dissertation would not have been possible. They were always there for me during the hard times, and always managed to help me whenever I was frustrated with my research.

Table of Contents

List of Figures xi

List of Tables xii

1 Introduction 1

1.1 Collecting Just-in-Time Statistics 3

1.2 Recommending Statistical Views 5

1.3 Challenges 8

1.4 Contributions and Dissertation Outline 9

1.4.1 Collecting and Maintaining Just-in-Time Statistics 9

1.4.2 Recommending Statistical Views 10

1.4.3 Dissertation Outline 11

2 Background and Related Work 12

2.1 Automated Collection of Statistics 12

2.1.1 Reactive Approaches 13

2.1.2 Proactive Approaches 15

2.2 Materialized Views 16

2.2.1 View Recommendation 17

2.2.2 View Exploitation 19

2.3 Statistical Views 21

2.3.1 Statistics on Query Expressions 21

2.3.2 Sample Views 23

2.3.3 Collecting Statistics on Statviews 25

2.3.4 Statistical vs. Materialized Views 26

2.4 The Principle of Maximum Entropy 29

3 Collecting and Maintaining Just-in-Time Statistics 31

3.1 Query-Specific Statistics 31

3.2 JITS Framework 33

3.2.1 System Architecture 34

3.2.2 Data Structures 35

3.2.3 Query Analysis 36

3.2.4 Sensitivity Analysis 38

3.2.5 Updating the QSS Archive 44

3.2.6 JITS Applicability 46

3.3 Experimental Evaluation 47

3.3.1 JITS for a Single Query 47

3.3.2 JITS for a Workload 49

3.3.3 Tuning the Sensitivity Analysis 52

3.4 Conclusions 53

4 Recommending Statistical Views 55

4.1 Problem Definition and System Overview 56

4.1.1 Key Insights 56

4.1.2 StatAdvisor Framework 62

4.2 Plan-Based Candidate Enumeration 64

4.2.1 Candidate Enumeration for a Query 64

4.2.2 Candidate Enumeration for a Workload 67

4.3 Benefit Estimation and Statview Selection 70

4.3.1 Benefit for a Single Query 71

4.3.2 Benefit for a Workload 72

4.3.3 Statview-Group Selection 73

4.4 Dependency on Database Engine 75

4.5 Experimental Evaluation 76

4.5.1 Setup 76

4.5.2 Candidate Enumeration 77

4.5.3 Overall Workload Performance 80

4.5.4 Comparison with Previous Work 84

4.6 Conclusions 85

5 Exploiting Statviews for Selectivity Estimation 87

5.1 Notations and Problem Definition 88

5.2 Estimation using Conditional Selectivity 89

5.2.1 Enumerating and Pruning Decompositions 92

5.2.2 Accuracy Estimation 93

5.2.3 Selecting the Best Estimate 98

5.3 Estimation using Maximum Entropy 100

5.3.1 Formal Definition 101

5.3.2 The Constrained Optimization Problem 102

5.3.3 Computing the Selectivity Estimate 106

5.4 Statview Matching Conditions 107

5.4.1 Matching for the Conditional Selectivity Approach 108

5.4.2 Matching for the Maximum Entropy Approach 110
5.5 Integration with the *PostgreSQL* Optimizer 111
5.6 Experimental Evaluation 112
5.6.1 Setup 112
5.6.2 Statview Matching Overhead 114
5.6.3 Workload Performance 114
5.6.4 Estimation Accuracy 116
5.7 Conclusions 117

6 Conclusions and Future Work 119

6.1 Conclusions 119
6.2 Future Work 120
6.2.1 Enhancing the Just-in-Time Statistics Functionality 120
6.2.2 Recommending “Generalized” Statviews 121
6.2.3 Statview Matching for Complex Query Expressions 122

Bibliography 128

List of Figures

2.1 View matching 20

2.2 View benefit 27

3.1 Database statistics 33

3.2 JITS architecture 34

3.3 Sample histogram 42

3.4 Histogram update 45

3.5 JITS benefit 50

3.6 Individual query performance 51

3.7 Sensitivity analysis threshold 52

4.1 Effect of statview-groups 59

4.2 *StatAdvisor* architecture 62

4.3 Important statviews 65

4.4 Candidate enumeration 66

4.5 Important statviews . . . 78

4.6 Convergence of candidate enumeration . . . 79

4.7 Execution plans for TPC-DS query 7 . . . 81

4.8 Workload performance . . . 83

4.9 *SITadvisor* vs. *StatAdvisor* . . . 85

5.1 Probability space for $|T| = 6$ and $N = \{1, 2, 3\}$ . . . 104

5.2 Maximum entropy solution . . . 105

5.3 Average matching overhead . . . 114

5.4 Workload Performance . . . 115

5.5 Absolute estimation error . . . 117

List of Tables

3.1 Statistics usage history 36

3.2 Table sizes 47

3.3 Compilation and execution times (in seconds) 48

4.1 *SITadvisor* vs. *StatAdvisor* 84

5.1 Computing environment for the experiments 113

Chapter 1

Introduction

Most modern applications deal with enormous amounts of data. Having to deal with such huge amounts of data is in itself a challenge. What makes this challenge even more complicated is that in many cases, this data is constantly changing and evolving. For example, relational databases that handle the data of day-to-day transactional applications often have tables with very high data change rates. It is not uncommon to even have temporary or volatile tables that get created from scratch and completely dropped over the course of one query workload. The constant change in data poses many challenges to how queries on this data are handled. We focus on optimizing structured queries over dynamic and constantly changing data sets.

Outdated Database Statistics. In relational databases, cost-based optimizers rely on a cost model to choose the best possible execution plan for a given query. The optimizer enumerates different execution plans, using different access paths, join orders, join methods and optimizations. The optimizer estimates the cost of these execution plans and chooses the plan with the least estimated cost. The accuracy of cost estimates is the main factor that affects the quality of the selected query execution plan. Cost estimates depend mainly on cardinality estimations of various sub-plans (intermediate results) generated during optimization. These cardinality estimates are computed using database statistics, which are

metadata maintained by the system that describe the underlying data. These statistics include the number of rows in a table, the number of distinct values in a column, the most frequent values in a column, and the distribution of data values (usually stored as a histogram).

With fast changing data, the stored statistics often become stale quickly as a result of data updates. Statistics are not incrementally updated during data manipulation because such incremental maintenance is prohibitively expensive. Traditional systems try to address this problem by periodically updating the stored statistics. This, however, is not particularly useful for tables with high data change rates, or temporary tables that get created and dropped during a workload. The presence of outdated statistics causes the optimizer to inaccurately estimate the costs of the operators in a query plan, which results in choosing a suboptimal plan. We discuss this issue in detail in Section 1.1.

Reducing Estimation Errors. Another challenge that faces the optimizer is how the database statistics are used in cardinality and cost estimation. In order to compute cardinality estimates of intermediate result sets from these statistics, query optimizers often employ some simplifying assumptions about the data and queries. For example, optimizers often assume that data is uniformly distributed over a given column (unless a histogram is available), and that query predicates are independent. However, these assumptions are usually incorrect, causing cardinality estimates to be off by orders of magnitude, leading to suboptimal execution plans.

In addition, it is often difficult for the optimizer to use the traditional statistics to estimate the cardinality and cost of query expressions that include complex 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 [26].

Example 1.1. Consider the following query Q_1:

```
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 often 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). 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 [26]. The estimation errors in both tables are further magnified as a result of the join predicate [36]. Reducing these estimation errors is the focus of Section 1.2.

This chapter starts by motivating the need for collecting just-in-time statistics in Section 1.1, and statistical views in Section 1.2.In Section 1.3 we list the challenges raised when attempting to address these research problems. We summarize our contributions and present the outline of this dissertation in Section 1.4.

1.1 Collecting Just-in-Time Statistics

As mentioned in the introduction, stored database statistics suffer from two main problems, which are a result of decoupling statistics collection and query processing: (a) the statistics

collection module has no knowledge of the queries posed to the system, which is why only general statistics are usually collected and stored, and why simplifying assumptions have to be made to be able to use these statistics; and (b) the stored statistics become outdated as a result of data updates. These two problems often result in estimation errors that can be orders of magnitude in size, and which in turn affect the quality of the selected execution plan.

It can be argued that if a query workload is known beforehand, then it is possible to analyze the whole workload, and collect all the needed statistics (including combined predicate selectivities, and histograms of skewed columns) at the beginning, thus ensuring that statistics are up-to-date and capture all correlations and non-uniform distributions. This approach only works for read-only workloads, or workloads with minor data updates. For workloads that include major data updates, the statistics collected at the beginning will quickly become outdated, and the staleness problem will surface once again.

In some cases, this can be detected and rectified using query feedback, by monitoring actual cardinality values during query execution, detecting the estimation errors, and reacting to those errors. This may involve re-optimizing the running query [37, 43], adjusting stored statistics to compensate for these errors in future queries [5, 50], or keeping multiple query plans and using the cardinalities monitored at run time to choose among these plans [11]. However, in some scenarios, query feedback may not be as useful. Examples include:

- Queries on tables with high data change rates
- Temporary tables that get created and dropped during a workload
- Ad hoc queries that are unlikely to be repeated

In all of these cases, query feedback fails to solve the problem of inaccurate statistics because of its learning curve. For tables with high data change rates, by the time the system adjusts to the errors in the statistics, the data is likely to have changed, and the

discovered adjustment is rendered useless. For temporary tables, the system is likely to have no statistics about them at all. Ad hoc queries are also problematic since the system might not have encountered them before, and thus has no idea how to compensate for the errors in their selectivities. Moreover, any information learned from such queries is useless as those queries might never be encountered again.

A brute-force approach to get accurate cost estimation would be to collect statistics on all data sources, and all the possible combinations of predicates in a given query before optimization. However, the problem with this approach is that (1) it is non-trivial to enumerate all statistics needed by the optimizer; (2) collecting all needed statistics for each query can be prohibitively expensive; and (3) it is hard to determine the most crucial statistics for the optimization process.

In our work, we propose an efficient approach to proactively determine, collect, and materialize Just-in-Time Statistics (JITS) for the currently optimized query. In contrast to earlier attempts, our approach employs a lightweight sensitivity analysis based on the query structure, the existing statistics and the data activity to identify the crucial statistics. The collected statistics are materialized and incrementally updated for future reuse.

1.2 Recommending Statistical Views

The inability to accurately estimate the output cardinality of complex query expressions triggered the idea of collecting statistics on views [26], also known as SITs – statistics on intermediate tables (or SQEs – statistics on query expressions) [13, 14]. In this dissertation, 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.

As seen in Example 1.1, cardinality estimation from base table statistics introduces several sources of errors, including the assumptions of independence and uniformity and the use of magic numbers to estimate the selectivity of predicates involving arithmetic expressions. However, given the query Q_1 from example 1.1, 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.

In the case of a read-only query workload that is known in advance (as opposed to ad hoc queries), it might be better to analyze the whole workload at once instead of using JITS with every query, in order to avoid the overhead of doing repeated work, and to exploit similarities between queries.

Example 1.2. Assume that there is a workload that contains the query Q_1 from Example 1.1, as well as the following query Q_2:

```
SELECT * FROM Car, Owner
WHERE Car.OwnerID = Owner.ID
AND Owner.city = 'Toronto'
AND Car.Price*(1-Car.Discount) > 4000
```

Note that both queries have predicates that include the expression *Price * (1-Discount)*. And even though the statview v_2 above provides accurate information for Q_1, it does not help with estimating the selectivity of the predicate in Q_2. Instead of creating another statview for the second query, it might be better to create the following statview:

```
v3: SELECT Price*(1-Discount) as dprice FROM Car
```

Collecting a histogram on the $dprice$ column in v_3 can provide accurate estimates for both queries.

In addition to exploiting similarities between queries, analyzing the whole workload can be performed as an offline process, to determine which statviews to create. These statviews can then be created once, and used every time that workload is executed. To the best of our knowledge, little work has been done to automate the process of deciding which statistical views to create given a SQL workload [13]. In addition, previous work did not study the interaction of multiple statviews when presented together to the query optimizer. In our work, we focus on recommending the most beneficial statviews for a workload, taking into account the interaction between statviews and their effect on query plans.

Statviews fall under the category of *multivariate statistics* (MVS) together with multi-dimensional histograms [45] and column-group statistics [34]. As shown in the earlier examples, these statistics provide more accurate selectivity estimates for groups of predicates, eliminating the need for the independence assumption. For a query with predicates $p_1, p_2, ..., p_n$, the optimizer has access to the following selectivity estimates:

- The individual selectivities $s_1, s_2, ..., s_n$ can be estimated from base table statistics.
- A limited collection of joint selectivities, such as $s_{1,2}$, $s_{3,5}$, and $s_{2,3,4}$ can be estimated using available statviews.

Using these statistics, the independence assumption is then employed to "fill in the gaps" in the incomplete information, e.g., the unknown selectivity $s_{1,2,3}$ can be estimated as $s_{1,2} * s_3$. This introduces a new problem: there may be multiple, non-equivalent ways of estimating the selectivity for a given set of predicates.

Example 1.3. Consider a query with the conjunctive predicates $p_1 \wedge p_2 \wedge p_3$. Base table statistics provide the optimizer with the selectivities s_1, s_2 and s_3 of p_1, p_2 and p_3 respectively. Suppose that there are statview that provide the selectivities $s_{1,2}$ and $s_{1,3}$ of $p_1 \wedge p_2$

and $p_1 \wedge p_3$ respectively. Using the available statistics and the independence assumption, the optimizer can estimate the combined selectivity of $p_1 \wedge p_2 \wedge p_3$ as either $s_{1,2,3} = s_{1,2} * s_3$, or $s_{1,2,3} = s_{1,3} * s_2$.

Any query plan that applies p_1 and p_2 first (e.g. using index ANDing) is likely to use the first estimate, while any plan that applies p_1 and p_3 first is likely to use the second estimate. This would result in an inconsistency if the two estimates are not equal, which is usually the case. Furthermore, there are potentially other choices, such as $s_1 * s_2 * s_3$ or, if $s_{2,3}$ is known, $s_{2,3} * s_1$. The choice of which estimate to use arbitrarily biases the optimizer toward choosing one plan over the other. Even worse, if the optimizer does not use the same estimate every time it is required, then different plans will be costed inconsistently, leading to incorrect comparisons and unreliable plan choices.

In our work, we implemented two methods of exploiting statviews in query optimization based on the work in [14] and [42]. The first method analyzes all possible ways of computing a selectivity estimate, and uses the one that gives the most accurate estimate. The second method relies on the *principle of maximum entropy* [31] to make use of all the available information while computing the estimate.

1.3 Challenges

There are multiple challenges associated with database statistics and query optimization. In this work, we focus on exploring the following challenges:

- *Determining needed statistics.* Determining which statistics are required to optimize a given query has to take several factors into account. These factors include the query structure (predicates and other constructs), what statistics are available, how old they are, how much the data has changed since these statistics were collected, how accurate the estimates obtained from these statistics can be, etc. Answering all

these questions becomes an even greater challenge if it has to be accomplished on-the-fly during query processing, since it has to be done in a lightweight manner that does not introduce too much overhead.

- *Finding beneficial statviews.* Deciding which statistical views are beneficial for a given query workload is not trivial. The effect of statviews (or statistics in general) cannot be simulated without using the traditional simplifying assumptions. Statviews also often interact with each other, and it is sometimes necessary to study them in groups rather than individually. In addition, the effect, or estimated benefit, of a statview cannot be determined by merely comparing the estimated cost of executing the workload with and without the statview present. The benefit of statviews has to take into account the special characteristics of statviews and how they are used in cardinality and cost estimation.

- *Exploiting statviews in selectivity estimation.* Once statviews are created, the optimizer needs to be able to "match" them with parts of any query that is being processed. If a match is successful, then the optimizer can use the statistics provided by the matched statview in estimating the output cardinality of the matched part of the query. Statview matching is different from traditional view matching (used with materialized views) because of how each type of views is used.

1.4 Contributions and Dissertation Outline

We present a summary of our contributions, and give the organization of the remainder of this dissertation.

1.4.1 Collecting and Maintaining Just-in-Time Statistics

We propose an efficient approach to proactively determine, collect, and materialize Just-in-Time Statistics (JITS) for a given query during query processing. Our key contributions are the following:

- We define the concept of *query-specific statistics (QSS)*, as opposed to general statistics (Section 3.1).

- We introduce a novel lightweight sensitivity analysis technique based on the query structure, the existing statistics and the data activity to identify the crucial statistics (Sections 3.2.3 and 3.2.4).

- We present a mechanism for materializing and incrementally updating the collected partial statistics for future reuse. Our approach integrates these partial statistics in a reusable form by maintaining maximum-entropy-based structures (Section 3.2.5).

1.4.2 Recommending Statistical Views

We present *StatAdvisor*, a system to automatically recommend statistical views (statviews) that are most beneficial for a particular SQL workload. Our key contributions are the following:

- We study the way statviews work, and present various key insights about their effect on query performance (Section 4.1).

- We introduce a novel iterative plan-based candidate enumeration algorithm based on the unique characteristics of statviews. The algorithm considers the possible dependency between multiple statviews in terms of their effect on the chosen execution plan (Section 4.2).

- We propose a benefit metric that takes into account the characteristics and effect of statviews. The system amortizes the benefit of the candidate statviews across the whole workload in order to get the final recommendations (Section 4.3).

- We demonstrate two different techniques to exploit statviews in query optimization. The first approach uses the most promising statviews (the ones thought to be the most accurate) to estimate a selectivity value, while the second approach incorporates information from all available and relevant statviews (Chapter 5).

1.4.3 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 gives an overview of related work from the literature. Chapter 3 describes our proposed framework for collecting and maintaining just-in-time statistics during query processing. Chapter 4 presents our proposed processing techniques recommending statistical views for a given SQL query workload. Chapter 5 presents the method we use for using statviews in cardinality estimation during query optimization. Finally, Chapter 6 gives our conclusions, discusses the limitations of our proposal, and lists a number of directions for future work.

Chapter 2

Background and Related Work

In this chapter, we review related work from the literature of query optimization, including automated collection of statistics (Section 2.1), materialized views (Section 2.2), and statistical views (Section 2.3). We also outline the principle of maximum entropy, which is relevant to our work, in Section 2.4.

2.1 Automated Collection of Statistics

Traditionally, it was the job of the user (or the database administrator) to determine which database statistics to collect and maintain. This type of labor-intensive activity has been a major component of the total ownership cost for a DBMS. Database designers have therefore expanded major efforts over the past decade to develop methods for automated statistics configuration and collection, that is, methods for the system to automatically decide which statistics to maintain, and when to collect and refresh these statistics. The decisions that need to be made include: the attribute sets on which to maintain joint statistics (also known as *multivariate statistics* or MVS), the number of frequent values, histogram bucket frequencies, the choice of bucket boundaries for histograms, etc. There have also been ef-

forts to make the statistics collection process more dynamic, gathering information during query execution so that the execution plan can be modified on the fly.

Most of the published work regarding the optimizer's dependence on statistics only addresses the problem of stale or outdated database statistics and their effect on query optimization. The problem of deciding which statistics to collect has not been thoroughly explored. 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. In contrast, proactive approaches try to predict, identify and possibly solve potential problems by doing additional work *before* query execution.

2.1.1 Reactive Approaches

The idea behind all of the reactive approaches is to piggyback the gathering of information on top of query execution, and exploit the information obtained from the query feedback for the benefit of future queries.

Adjusting Statistics

The initial version of the *Learning Optimizer (LEO)* in *IBM DB2* [50] monitored actual cardinality values along the edges of the query plan. These values were compared to the estimates used by the optimizer. The error is used as a correction factor for the optimizer's selectivity estimates (to be used by future queries). However, the LEO approach has evolved into a method for automated statistics configuration. As each predicate is evaluated during query execution, a query feedback record, which contains the predicate's actual selectivity and the optimizer's selectivity estimate, is created. These query feedback records are stored into the *query feedback warehouse (QFW)*, and later used for a variety of statistics-related tasks.

A slightly different approach [5] uses the monitored error to trigger statistics collection if the error exceeds a certain threshold. However, the current query still suffers from incorrect estimates. The problem is further magnified in systems with long-running ad hoc unrelated queries, in which case the adjustment in statistics is unlikely to be used by future queries.

The notion of using query feedback to update the existing statistics has received considerable attention. STHoles [15] introduced a novel data structure – a histogram whose buckets can have holes. This histogram gets updated whenever information from query feedback is available. The ISOMER method [49] builds on that approach, but uses the principle of maximum entropy to approximate the true data distribution by the simplest distribution that is consistent with all the currently valid feedback without any further assumptions. Similar approaches for piggybacking statistics collection with query execution are discussed in [6, 12, 55]. More details on the principle of maximum entropy are given in Section 2.4.

Query Re-optimization

A different reactive approach reacts to observed estimation errors by re-optimizing the current query as it is being processed. The approach proposed in [37] inserts statistics-collection operators at different points in the query execution plan. The purpose of these operators is to monitor the actual cardinality and compare it to the optimizer's estimate. If the estimation error exceeds a certain threshold, the system decides that the current execution plan is suboptimal. In that case, the system tries to optimize the execution, either by changing the resource allocation or by finding a new execution plan based on the newly observed statistics and re-executing the query using the new plan.

In *Progressive OPtimization (POP)* [43], the optimizer chooses a plan based on the existing statistics. The optimizer then calculates a validity range for the cardinality at each intermediate result, that is, the range of values in which the current plan is still the optimal one. In addition, statistics collection operators are inserted in a manner similar to the approach in [37]. During query execution, actual values are monitored. If an actual

cardinality value is outside the validity range of the corresponding operator, the optimizer decides that the current plan is not optimal anymore, and the query is re-optimized.

However, it is hard to monitor actual cardinalities during query execution and use them for possible re-optimization without blocking the execution pipeline. Furthermore, the decision to re-optimize raises a question of whether to reuse the partial results that have been obtained already or to start execution from the beginning with the new plan. Changing the execution plan adaptively during query execution has also been the focus in [9], even though this change comes as a response to fluctuations in resource availability, rather than to estimation errors.

2.1.2 Proactive Approaches

Babu et al. [11] proposed an approach that is partly proactive but mostly reactive. This approach is based on the possible error in the cardinalities at every edge of the query plan. At each operator, the system computes the possible range of values for the input relation(s) to this operator, getting a *bounding box*. The system maintains three alternative "switchable" sub-plans for each operator; ones that are optimal at the lowest, middle, and highest points in the bounding box. During execution, the approach is reactive; the system detects the actual cardinality and chooses one of these sub-plans accordingly if the value is inside the bounding box; otherwise, it re-optimizes the query. The main problem is that the three maintained sub-plans do not necessarily cover the whole spectrum of possible sub-plans, i.e., there could be other plans that are better than the selected three plans for a particular input cardinality value. Furthermore, this approach depends on the ability to produce meaningful intervals around cardinality estimates, which is a hard problem by itself.

To the best of our knowledge, the only work that addresses the issue of choosing which statistics to collect before query execution is the MNSA (Magic Number Sensitivity Analysis) approach proposed in [18]. It includes a technique to perform sensitivity analysis in order to decide which statistics to collect so that the optimizer will have enough information to optimize that query. The idea is to check whether the currently available set of statistics

is *sufficient* or not. If not, then collect the *most important statistic*, and then repeat the check again until the available statistics are sufficient. The decision of whether the current set of statistics is sufficient or not is taken by invoking the optimizer twice. In the first invocation, all unknown selectivities are set to a very small value $\varepsilon > 0$. In the second invocation, all unknown selectivities are set to a large value $1 - \varepsilon$. If the estimated costs of the two generated plans are within $t\%$ of each other (for a predefined value of t), the current set of statistics is said to be sufficient. If not, the system identifies the *most important* statistic by calling the optimizer again to get an execution plan based on the current set of statistics, then comparing the estimated costs of the operators in the plan, assuming that expensive operators are associated with important statistics.

However, this approach has several shortcomings. It requires multiple calls to the optimizer for every statistic, which can be very time-consuming especially for complex queries. In addition, although this approach can be particularly useful if most of the tables involved in the query have up-to-date statistics, it would not be as useful if most of the statistics are outdated. This is because it decides the importance of statistics based on the estimated operator costs in the execution tree that is already built using inaccurate information.

2.2 Materialized Views

In an RDBMS, a view is a *virtual* table representing the result of a database query. Whenever an ordinary view's data is queried or updated, the DBMS converts these operations into queries or updates against the underlying base tables. A *materialized view* takes a different approach in which the query result is physically stored as a concrete table that may be refreshed from the original base tables from time to time. If a user issues a query over a materialized view, its results can be directly returned instead of having to be recomputed from the base tables, which helps speed up query execution. Furthermore, if the user poses a query over base tables, with similar predicates to what is used in a materialized view, the optimizer can choose to use the materialized view instead of the base table, thus potentially avoiding to redo complex computations. This enables much more efficient access, at the

cost of some data being potentially out-of-date. In addition, materialized views are treated just like base tables. Anything that can be done to a base table can be done to a materialized view, most importantly building indexes on any column, enabling drastic speedups in query execution time.

Building and maintaining query performance enhancers, such as database indexes and materialized views has been extensively studied. Query processing time can be improved by orders of magnitude through judicious use of materialized views. In order to make full use of materialized views, it is required to address three issues [28]:

- **View design:** determining which views to materialize
- **View exploitation:** using the views to speed up query processing
- **View maintenance:** efficiently updating the views when the base tables are updated

In this section, we focus on the first two issues since they are the most relevant to our work. We outline the recent work in both directions in the next two subsections.

2.2.1 View Recommendation

A considerable amount of work has been done to automate the process of recommending which indexes or materialized views to construct. The *Design Advisor* in *IBM DB2* [51, 56] analyzes a given workload. For each query in the workload, the advisor builds a list of candidate views and indexes. These candidates are generated using some heuristic rules based on the structure of the query. In order to estimate the benefit of each candidate to the query under consideration, the advisor simulates the existence of all the candidates (by creating entries in the system catalog without creating the actual views, in addition to estimating statistics about these candidates using basic cardinality estimation techniques). The advisor then calls the query optimizer for the given query. The optimizer chooses the best execution plan for the query taking into consideration all the virtual candidates.

The chosen plan might include some of those candidates. Those candidates are assigned a benefit score (based on the saving in the estimated execution time) and a cost (based on their size). Candidates that are not in the chosen plan are pruned. This process is repeated for every query in the workload, and the benefit scores of the candidates are aggregated across all queries. Finally, a knapsack algorithm is used to select among all the candidates to achieve the maximum benefit while satisfying the space constraints.

A slightly different approach is used in *Microsoft SQL Server* [2, 8]. First the system finds the *"interesting table subsets"*, that is, sets of tables that are involved in multiple queries and/or queries of estimated high execution cost. To find these sets, the system uses a scoring mechanism, where each subset of the tables in each query is assigned a score. This score is directly proportional to the estimated cost of the query, as well as the sizes of the tables involved. If a subset occurs in more than one query in the workload, the score of that subset is aggregated. Interesting table subsets are those subsets whose score across the whole workload exceeds a predefined threshold. Once those sets are identified, for each such set in each query, the system generates some candidate views and indexes. The optimizer is then used in a way similar to that in [51, 56] to get the query-level recommendations, and to compute the benefit of each candidate. Subsequently, the system attempts to generate more candidate views by merging similar views that are recommended for different queries. A given pair of views (referred to as the parent views) are used to generate a new view (called the *merged view*) if these conditions are met: (a) all queries that can be answered using either of the parent views should be answerable using the merged view, and (b) the cost of answering these queries using the merged view should not be significantly higher than the cost of answering the queries using the parent views. Finally, a greedy algorithm is used to determine the final workload-level recommendations based on the estimated benefit and cost of each candidate. Generating merged views that can be used by multiple queries has also been an ongoing research problem [39].

2.2.2 View Exploitation

Using materialized views to enhance query performance is mainly based on *view matching* (or *query matching*). View matching is performed on the internal query representation (e.g. the Query Graph Model - QGM [32]) before plan enumeration. In such models, the query is usually represented as a tree with nodes (often referred to as query blocks) representing base tables, selections, joins, and aggregations, and edges representing data flow. Each block specifies what attributes are produced, and what predicates (if any) are used to filter the outputs. This representation is different from execution plans, since it does not show the join order, join methods, or physical operators to be used.

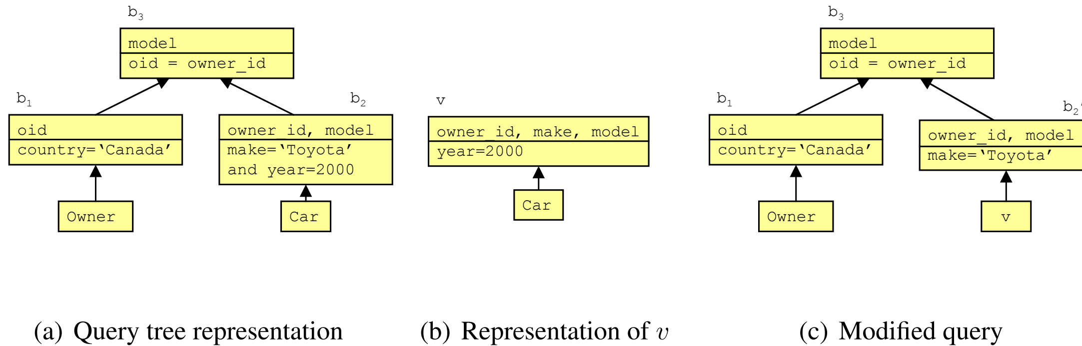
Example 2.1. Consider the database tables: *Owner (oid, name, country)* and *Car (cid, make, model, year, owner_id)*. Also consider the query:

```
SELECT model FROM Car, Owner
WHERE oid = owner_id
AND country = 'Canada'
AND make = 'Toyota'
AND year = 2000
```

Figure 2.1(a) shows the tree representation of this query. Block b_1 represents the selection on the *Owner* table, b_2 represents the selection on the *Car* table, and b_3 represents the join.

Since views are also based on queries, they are also represented in the same way. In view matching, the optimizer tries to match a query block with any of the existing materialized views. If a successful match is found, then another query tree is created, using the view instead of the base table(s), and the optimizer is allowed to enumerate plans for both trees, and choose among them based on the estimated cost. A major challenge is being able to efficiently check the query for matching with the existing views. The problem becomes even more challenging as the number of materialized views in the system increases, and as queries get more complex (with sub-queries, and arbitrary expressions).

View matching is based on *coverage* or *subsumption*; a materialized view is considered an exact match to a certain query block if the view produces the same tuples and attributes



(a) Query tree representation (b) Representation of v (c) Modified query

Figure 2.1: View matching

that are produced by that block [54]. In case of non-exact matches, where the view produces more tuples and/or attributes than the query block, a compensation block is added to output only the desired tuples and attributes. This compensation is usually in the form of applying additional predicates, projections, and/or aggregate functions to get only the desired results.

Example 2.2. Given the tables and query in Example 2.1, assume that there exists a materialized view v defined as: $v =$ `SELECT owner_id, make, model FROM Car WHERE year=2000`. Figure 2.1(b) depicts the tree representation of v. This view can be used in place of the car table for the purpose of the given query, since it covers the tuples and attributes needed by the query. However, a compensation predicate (block b'_2 in Figure 2.1(c)) needs to be added.

The view matching problem has received considerable attention. Zaharioudakis et al. [54] proposed a solution to the problem based on the Query Graph Model (QGM) [32] used in *IBM DB2*. Their solution exploits the QGM structure to match multi-block queries with multi-block views, including arbitrary aggregate functions and arithmetic expressions. The approach considers different cases for exact and non-exact matches, and computes the compensation expression in each case.

Goldstein and Larson also addressed the same problem in [28]. Their view matching algorithm handles views with selections, projections, joins and a final group-by (SPJG). For each view, the algorithm considers equivalence among columns (as implied by equality

predicates), and uses this information to test whether or not the view contains all the rows needed by the query. The work in [28] also addresses the performance issues caused by the large number of materialized views in the system. To overcome this problem, the approach includes an indexing mechanism that quickly narrows the search down to a small set of candidate views on which view matching is applied.

2.3 Statistical Views

Most query optimizers use a cost model to choose the best query execution plan. The quality of the selected plan depends mainly on the accuracy of the cost estimates computed by the optimizer, which in turn depend on the cardinality estimates of intermediate results. Traditionally, database systems only maintain statistics on base tables and their attributes. The optimizer has to make some simplifying assumptions in order to estimate the cardinality of an intermediate query expression using the base table statistics. The assumptions made are often unrealistic. Examples of these assumptions include predicate independence and data uniformity. As seen in the introduction (Example 1.1), the simplifying assumptions made by the optimizer can cause the cardinality estimates (and cost estimates) to be off by orders of magnitude, which often results in the optimizer choosing a sub-optimal execution plan.

This problem inspired the idea of maintaining statistics not only on base tables, but also on intermediate query expressions. In the following subsections, we outline two different approaches for computing statistics on intermediate results (query expressions).

2.3.1 Statistics on Query Expressions

Bruno and Chaudhuri [13, 14] introduced *SITs* (statistics on intermediate tables), also known as *SQEs* (statistics on query expressions). In their work, a SIT is a statistic (usually a histogram) over a column in the result of a query expression. A SIT can be seen as a special case of a statview, which has a single output column. The work in [13, 14] extended

traditional optimizers to exploit statistics built on expressions corresponding to intermediate nodes of query plans. SITs are similar to materialized views, except that the result of the view is not materialized, but only used for statistics collection and then discarded. Only the statistics are maintained by the system. The major challenges of this approach are: (a) determining for which of the many sub-expressions of an SQL workload the system should collect SITs, and (b) ensuring that the query optimizer is able to exploit SITs if they exist for some sub-expression of a given query.

To determine which SITs to collect, the approach in [13] can be viewed as three main steps:

1. Finding *"interesting"* predicates in a given query
2. Determine the generating query for each SIT
3. Aggregating query-level recommendations into workload-level recommendations

To find the interesting predicates in a query, the work in [13] builds on the MNSA technique [18]. For each filter predicate, the system tries to determine the sensitivity of the query cost to this predicate. This is accomplished by using two estimation techniques *Min* and *Max*, which make extreme assumptions about the distribution of values in the attribute involved in the predicate. The two estimation techniques yield two sets of cardinality estimates. The optimizer is called twice (once for each set of estimates) returning two execution plans with two estimated execution costs. The difference between the estimated costs is interpreted as the sensitivity of the query to this particular attribute; If the estimated cost difference is small, building a SIT for the attribute in question will have little effect on the query.

To determine the generating query for a SIT, the *Min* and *Max* estimation techniques are used to propagate the extreme assumptions from the predicate in question to the whole query result. A score is assigned to each partial join (candidate SIT) based on how much it contributes to the uncertainty in the final result. Scores are also multiplied by the difference in estimated cost, so that SITs used in more expensive queries would be deemed more

beneficial. The scores are then aggregated across the workload (for SITs that are used by multiple queries), and a greedy selection algorithm is used to select the SITs with the maximum scores that can fit within a space constraint.

The approach in [13] suffers from several shortcomings: Predicates are considered one at a time, assuming independence, which does not solve problems arising from correlation of predicates. Also, comparing the costs computed using different sets of statistics (using *Min* and *Max*) is not indicative of importance: A large cost difference can be the result of just the different statistics, even though the execution plan did not change. In addition, for each filtering predicate in the query, the query is optimized twice, which can be very expensive. Last but not least, SITs are recommended one at a time, not considering possible dependencies between them. In real situations, the availability of one statistic might not be beneficial unless another particular statistic also exists.

During query evaluation, the work in [13] uses standard view matching techniques to match the existing SITs with all subexpressions of a given query, in order to decide which views to use for estimating the cardinalities of these expressions. The authors extend that approach in [14] by introducing the concept of *Conditional Selectivity*, which enables the optimizer to match the query with several SITs at the same time, which would have been missed using the traditional query matching techniques. More details about the conditional selectivity concept is given in Chapter 5.

2.3.2 Sample Views

Larson et al. [38] proposed using *Sample Views* for cardinality estimation. Sample views are also similar in concept to materialized views except that they contain only a random sample of the view result. Given a query that is being optimized, view matching techniques are used to determine which sample views can be used, and probe queries are issued against these views to compute the required cardinality estimates. To ensure that the accessed records from the view are truly a random sample, a method called *sequential sampling* is used. When creating the view, an additional column, called `RAND` , is added to the view.

For each tuple in the view, the `RAND` attribute is assigned a random value drawn from a uniform distribution in the range [0, `MAXRAND`], where `MAXRAND` is a relatively small integer value (maximum 1000). The sample is then stored in sorted order on the `RAND` column. Thus rows with the same `RAND` value are clustered together. For a particular value of the `RAND` attribute, the group of tuples with that value can be seen as a random sample. When issuing probe queries against the sample, the rows are scanned sequentially until the end of a cluster (when the `RAND` value changes). The set of rows scanned is a valid random sample. Sequential sampling only processes enough rows to compute a sufficiently accurate estimate (with guarantees on the standard error).

To maintain the samples, query feedback is used to report back the actual values of the estimates computed from the views. If the feedback shows that a certain view has become stale (the reported value differs from the computed estimate by more than a predefined error threshold), then a refresh request is triggered for this particular view, after which the view enters a "refresh pending" mode. In this mode, it can still be used for query optimization, but the optimizer is aware that estimates obtained from that view may not be accurate. When the load on the system allows, the sample is dropped, and a new sample is recollected. For some views, there might be insufficient feedback, due to the following reasons:

- View is not used during query optimization
- Expression estimated by the view does not appear in the final plan
- Expression appears in the final plan, but it does not have a valid count (e.g. because of early termination)

In these cases, there is not enough information to determine whether a view needs to be refreshed. Therefore, in order to safeguard against stale views, the system (or DBA) can either issue a forced update, or some artificially generated and specifically tailored guard queries just to get feedback for these particular views.

2.3.3 Collecting Statistics on Statviews

The simplest 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 $v_2 = \sigma_{p_2}(A)$, where p_1 and p_2 are arbitrary combinations of predicates on table 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 [7, 17, 44]. To overcome this problem, join synopses [7] can be used. The join synopsis for table A is built as follows [10]:

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. Note that join synopses can only be used with non-cyclic joins involving foreign keys.

2.3.4 Statistical vs. Materialized Views

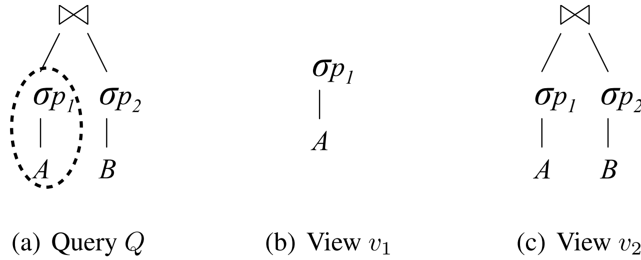
Statistical views are similar in concept to materialized views except that materialized views contain pre-computed data, while statistical views are used only for cardinality estimation, not for query evaluation. Exploiting materialized or statistical views is based on *view matching*. The optimizer 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).

However, 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

The difference in how to evaluate the benefit of statviews and materialized views is best illustrated by the following example:

Example 2.3. Figure 2.2(a) 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, as shown in Figures 2.2(b) and 2.2(c). 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



(a) Query Q (b) View v_1 (c) View v_2

Figure 2.2: View benefit

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.

This example shows that the benefits of materialized views and statviews are evaluated differently. Hence, the benefit metrics used for materialized views cannot be used for statviews. In Section 4.3, 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 [54]. 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 [54] 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 2.4. Consider the following query Q and views v_1, v_2 and v_3:

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

v_1:
```
SELECT DISTINCT R.e,
S.c, S.f FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
```

v_2:
```
SELECT R.e, S.c,
AVG(R.d) FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
GROUP BY R.e, S.c
```

v_3:
```
SELECT R.e, S.c, S.f
FROM R, S
WHERE R.a = S.a
```

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 [22, 49]. 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 4.4.

The aforementioned differences between statviews and materialized views, as well as other special properties that we discuss in Section 4.1, warrant the development of a dedicated advisor that takes these special characteristics into account.

2.4 The Principle of Maximum Entropy

The *maximum entropy* (ME) principle [31] models all that is known and assumes nothing about the unknown. It is a method for analyzing the available information in order to determine a unique epistemic probability distribution. Given a probability distribution $\overline{q} = (q_1, q_2, ..., q_n), \sum_{i=1}^{n} q_i = 1$, Information Theory [48] defines a measure of uncertainty called entropy:

$$H(\overline{q}) = H(q_1, ..., q_n) = -\sum_{i} q_i \cdot log(q_i) \tag{2.1}$$

Each value q_i may be seen as the probability of the ith outcome of a probabilistic experiment or the probability of the ith possible value taken on by a finite discrete random variable. As already mentioned, the entropy can be viewed as the uncertainty resulting from such probability distribution. Keeping this in mind, entropy has some interesting properties [31]:

1. If $\overline{q}$ has only one component which is different from zero (i.e., equal to 1) then $H(\overline{q}) = 0$. This makes sense, as there is no uncertainty if there is only one possible outcome.

2. $H(q_1, ..., q_n) \leq H(1/n, ..., 1/n)$, with equality if and only if $q_i = l/n, i = 1, ..., n$. This simply means that the entropy (or uncertainty) is maximized if all possible outcomes are equally likely. Thus, of all possible probability distributions, a uniform distribution yields the highest entropy.

3. For s random variables with arbitrary finite range, $H(X_1, ..., X_s) \leq H(X_1) + ... + H(X_s)$, with equality if and only if $X_1, ..., X_s$ are globally independent. This means that entropy of multiple random variables is maximized if these variables are independent.

The ME principle prescribes selection of the unique probability distribution that maximizes the entropy function $H(\overline{q})$ and is consistent with respect to the known information. Entropy maximization without any additional information uses the single constraint that the sum of all probabilities is equal to one.

Consider the above properties in the context of query optimization. If the optimizer does not use multi-variate statistics (MVS), then it actually estimates the selectivities of conjunctive queries according to the ME principle: the optimizer assumes uniformity when no information about column distributions is available, and it assumes independence because it does not know about any correlations.

By integrating the more general concept of maximum entropy into the optimizer's selectivity model, we thereby generalize the concepts of uniformity and independence. This enables the optimizer to take advantage of all available information in a consistent way, avoiding inappropriate bias towards any given set of selectivity estimates.

Chapter 3

Collecting and Maintaining Just-in-Time Statistics

In this chapter, we describe the techniques we proposed in [22] to proactively determine, collect, and materialize Just-in-Time Statistics (JITS) during query processing. We start by defining what we refer to as Query-Specific Statistics in Section 3.1. We discuss the details of our JITS framework in Section 3.2. We present our experimental study in Section 3.3 and summarize this chapter in Section 3.4.

3.1 Query-Specific Statistics

As mentioned in the Introduction (Chapter 1), cost-based query optimizers often make some unrealistic assumptions while using *general database statistics* for cardinality and cost estimation. These assumptions (e.g. independence and data uniformity) often do not hold, and usually yield high estimation errors. This raises the need for *Query-Specific Statistics (QSS)*, which take into consideration the predicates and the values used in a particular query. QSS allow the optimizer to accurately estimate the cost of different execution (sub)plans, while making fewer assumptions on the underlying data.

Example 3.1. Consider the following query:

```
SELECT E.name
FROM Employee E, Company C
WHERE E.compid = C.id
AND C.name = 'Microsoft'
AND E.salary > $100k
AND E.age < 30
```

A possible QSS for this query is the combined selectivity of the two predicates $salary > \$100k$ and $age < 30$, which would allow the optimizer to accurately estimate the number of satisfying tuples, based on which it can choose the best access path for the $Employee$ table, and the best join method. However, since most systems do not maintain such statistics, the optimizer has to rely on general single-column statistics to compute these estimates. Even with the existence of histograms on the age and $salary$ columns, the optimizer usually assumes uniformity within each histogram bucket. Furthermore, since it is impossible to maintain multi-dimensional histograms for all possible combinations of columns, the optimizer may assume independence to compute the joint selectivity of the two predicates. These assumptions often lead to large errors in cardinality and cost estimation. The problem is further magnified with more complex queries that involve large numbers of predicates and joins [36].

Note that predicate selectivities, in addition to being query-specific, can also be *plan-specific*; the selectivity of a particular predicate can be useful for costing a certain plan but useless for costing another plan that evaluates the same query.

Example 3.2. Consider a query that contains a join of 3 tables A, B, and C. In one possible plan, the join order is $(A \bowtie B) \bowtie C$. This plan needs the selectivity of the join predicate of $A \bowtie B$. Another plan with a different join order $A \bowtie (B \bowtie C)$ would need the selectivity of the join predicate of $B \bowtie C$. Hence, in general, collecting all needed statistics would involve collecting the selectivity of all possible join predicates and other plan-specific statistics, which can be prohibitively expensive.

As shown in Figure 3.1, QSS can be viewed as a compromise between: (1) general statistics currently collected by query optimizers, where multiple unrealistic assumptions

	General Statistics	Query-Specific Statistics (QSS)	Plan-Specific Statistics
Usability	Most Queries		Some plan(s) on a particular query
Accuracy	Low		High
Collection Cost	Low		High
Examples	Base statistics on tables and columns		• Combined selectivity of multiple predicates • Join selectivity of different table combinations

Figure 3.1: Database statistics

are made to generate the required statistics for cost estimation; and (2) plan-specific statistics that can be directly used in cost estimation, eliminating the need for assumptions but suffering from prohibitively expensive collection cost. QSS takes the middle ground between the two types of statistics in terms of usability, accuracy and collection cost.

A system that collects and exploits QSS in query optimization needs to address the following issues: (1) which QSS to collect among a large number of possible candidates; and (2) how to efficiently materialize the collected QSS for later reuse. Section 3.2 describes the JITS framework, and how it tackles these issues.

3.2 JITS Framework

In this section, we describe JITS, a system for proactively collecting query-specific statistics during query compilation. We give the overall architecture of JITS, describe the structure of the QSS archive, and give details of our implementation of the various modules.

3.2.1 System Architecture

Figure 3.2 depicts the architecture of JITS. Entities in dotted lines already exist in current query engines, while entities in solid lines are new JITS modules.

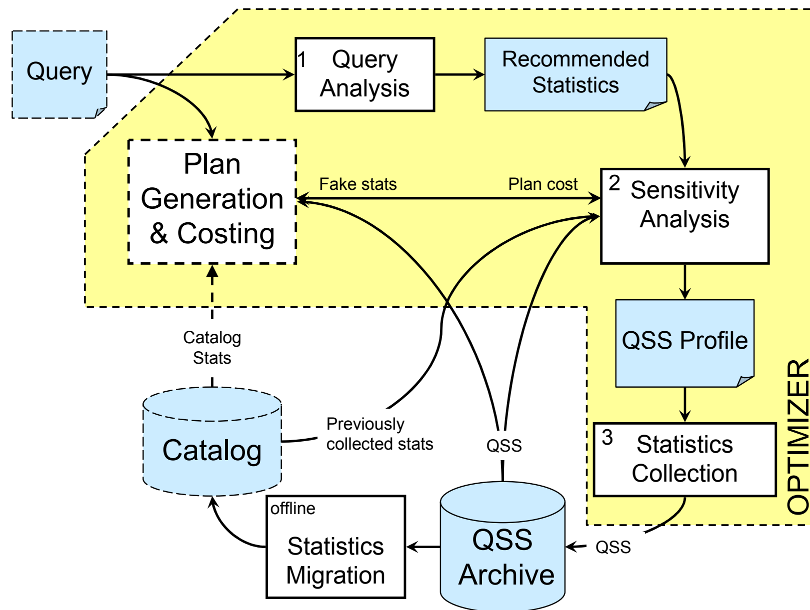


Figure 3.2: JITS architecture

The *Query Analysis* module analyzes the query structure, after parsing and rewriting, to determine all relevant statistics, and generates a list of candidate statistics. The *Sensitivity Analysis* module processes the candidate statistics to decide the most crucial statistics to collect. Our implementation examines the query, existing statistics, as well as the history of data activity (e.g., frequency of updates and deletes on a particular table). In general, the sensitivity analysis module can use any more sophisticated technique, e.g., by incorporating the planning module to determine the sensitivity of the query to a particular statistic [18]. The output of the sensitivity analysis is the set of statistics that needs to be collected. We

refer to this set as the *QSS profile*. The *Statistics Collection* module collects the required statistics, and uses them to update the QSS archive. The Plan Generation and Costing module uses the information in the QSS archive and the system catalog to select an execution plan. The information in the QSS archive can be used to periodically update the system catalog using the *Statistics Migration* module.

3.2.2 Data Structures

The JITS framework maintains and uses the following data structures:

QSS archive: This is a repository of adaptive single- and multi-dimensional histograms. Categorical and character data types can be represented as numerical values using a mapping function to allow for interpolation. We elaborate on the details of the histograms and their update strategy in Section 3.2.5.

UDI counter: For each base table, we maintain a counter that encapsulates the number of *updates*, *deletions* and *insertions* that took place since the last statistics collection on this table. We use the UDI counter as an indication of the change in the data. This is used by the sensitivity analysis module (Section 3.2.4).

StatHistory: The query optimizer can usually estimate the selectivity of conjuncts of predicates such as $sel(p_1 \wedge p_2 \wedge p_3 \wedge p_4)$ by using partial selectivities such as $sel(p_1)$, $sel(p_2 \wedge p_3)$, and $sel(p_2 \wedge p_3 \wedge p_4)$ [42]. We maintain a history of the usage of statistics in selectivity estimation. This history can be used to evaluate the effectiveness of the optimizer's assumptions in estimating combined selectivities from partial statistics. Each *StatHistory* entry corresponds to the optimizer using a particular set of statistics to estimate the selectivity of a group P of conjunctive predicates. Each entry is of the form $(T, colgrp, statlist, count, errorfactor)$, where:

- T is the table to which the predicate group P belongs
- $colgrp$ are the columns in the predicate group P

Table 3.1: Statistics usage history

T	colgrp	statlist	count	errorfactor
T_1	(a, b, c)	$\{(a, b), (c)\}$	5	0.4
T_1	(a, b, c)	$\{(a), (b, c)\}$	2	0.7
T_1	(a, b, c)	$\{(a, b, c)\}$	10	0.98
T_1	(a, b, d)	$\{(a, b), (d)\}$	4	0.8

- $statlist$ is a set of statistics that were used to estimate the selectivity of P
- $count$ is the number of times that the statistics in $statlist$ have been used to estimate the selectivity of P
- $errorfactor$ is the ratio between the estimated selectivity and the actual selectivity of P

The $errorfactor$ value is usually provided by a feedback system that monitors the actual selectivities and compares them to the optimizer's estimates (e.g., LEO [50]). Table 3.1 gives a sample of the stored $StatHistory$. The first entry states that the selectivity of a predicate group involving columns a, b, and c in table T_1 (e.g., *a=5 AND b>10 AND c<100*) has been estimated using combined statistics on columns a and b and statistics on column c, and that this scenario has happened 5 times. The ratio between the estimated selectivity and the actual monitored selectivity is 0.4 (average over all 5 occurrences). Note that *StatHistory* does not store the predicate groups themselves, but only the columns referenced in these predicates. For example, the first entry in Table 3.1 refers to all instances where statistics on (a, b) and statistics on c have been used to estimate the combined selectivity of a predicate group referencing columns a, b and c.

3.2.3 Query Analysis

The *Query Analysis* module determines which statistics are relevant to the query, regardless of whether or not they should be collected. These statistics can be classified as: (1) table

Algorithm 3.1 Query analysis

```
1: function QUERYANALYSIS(Q)
2:     PG ← ϕ
3:     B ← set of query blocks in Q
4:     for all b ∈ B do
5:         P ← set of predicates in b
6:         T ← set of tables involved in b
7:         for all t ∈ T do
8:             P_t ← {p|p ∈ P, p is local on t}
9:             PG ← PG ∪ {P_t}
10:        end for
11:    end for
12:    return PG
13: end function
```

statistics (e.g., number of rows), which are needed for every table involved in the query, and (2) column statistics, which basically include the selectivities of predicates or groups of predicates. Since table statistics are usually collected and maintained by most systems, we focus on statistics related to single predicates and predicate groups.

The query analysis (Algorithm 3.1) takes as input a query Q and returns the set PG of candidate predicate groups on which statistics are needed in order to optimize Q. Each element in PG is a group of predicates that appear in the query. The algorithm analyzes the query by examining its internal structure after parsing and rewrite. Since the aim of QSS is to be directly used by the optimizer, the algorithm collects predicate groups per query block (SPJ block), since most optimizers perform intra-block optimization. For every query block b, the algorithm finds all the predicates belonging to the same table, and adds these predicates (as a group P_t) to final candidate list PG.

Example 3.3. Consider the following query:

```
SELECT *
FROM A, B
WHERE A.a1 = B.b1
AND A.a2 < 10
AND (A.a3 = 5 OR A.a3 = 7)
AND B.b2 > 20
AND not EXISTS (
  SELECT * FROM A as AA, C WHERE AA.a1 = C.c1
  AND AA.a2 > 0 AND AA.a4 = A.a4)
```

This query contains two query blocks, corresponding to the main query and the subquery. While processing the outer block, the algorithm finds the following predicate groups:

- `A.a2 < 10 and (A.a3 = 5 or A.a3 = 7)` (corresponding to table A)
- `B.b2 > 20` (corresponding to table B)

The inner query block produces the following predicate group:

- `AA.a2 > 0` (corresponding to table A)

These three predicate groups are the candidate QSS for this query.

3.2.4 Sensitivity Analysis

Statistics collection during query compilation is an expensive process. Collecting all statistics recommended by the query analysis module is not always necessary. Therefore, it is crucial to decide which statistics are necessary to collect.

Algorithm 3.2 Sensitivity analysis

```
 1: procedure SENSITIVITYANALYSIS(Q, PG)
 2:     T ← set of tables involved in Q
 3:     for all t ∈ T do
 4:         PG_t ← {g|g ∈ PG, g is local on t}
 5:         if ShouldCollectStats(t, PG_t) then
 6:             Mark t for statistics collection
 7:             for all g ∈ PG_t do
 8:                 if ShouldMaterialize(g) then
 9:                     Mark g for materialization
10:                 end if
11:             end for
12:         end if
13:     end for
14: end procedure
```

The sensitivity analysis (Algorithm 3.2) takes as input the query Q and the list of predicate groups recommended by the query analysis module. The algorithm makes use of two other subroutines that are explained in detail in the following sections: $ShouldCollectStats$ and $ShouldMaterialize$. The subroutine $ShouldCollectStats(t, PG_t)$ (Algorithm 3.3) determines if statistics should be collected on a table t based on the candidate statistics PG_t. Ideally, the sensitivity analysis evaluates the "importance" of each candidate statistic. We adopt a simplification heuristic that decides on all the statistics PG_t of a table as a single unit. The rationale is that most of the cost of collecting the statistics is in the sampling process. Once a table is sampled, it is relatively cheap to collect the selectivities of all predicate groups that belong to this table.

In some cases, some of the collected statistics might not be useful for future queries, and storing them would be a waste of space, especially if they involve creating new QSS histograms (more details on updating the QSS histograms are given in Section 3.2.5). Therefore, to meet space constraints, it is important to decide which statistics are more likely to be useful in the future. The subroutine $ShouldMaterialize(g)$ (Algorithm 3.4) determines

if a certain predicate group g should be materialized. The details of the two subroutines are explained next.

1. Determining Crucial Statistics to Collect

Deciding whether or not to collect statistics on a particular table is mainly based on evaluating two metrics:

- s_1 reflects the accuracy of currently existing statistics on this table; and
- s_2 reflects the data activity on the table.

Each of the two metrics can be viewed as a value ranging from 0 to 1; where 0 means that no statistics collection is needed and 1 meaning that statistics must be collected. The "importance" of a particular statistic can be computed as a function of s_1 and s_2.

Computing these scores is described in Algorithm 3.3. To compute s_1 based on a particular predicate group g, the algorithm fetches all the $StatHistory$ entries that refer to this group (line 3). For example, if this group is *(a=5 AND b>10 AND c<100)*, and the history is as shown in Table 3.1, then H will have all the entries which have the colgrp (a, b, c), i.e. the first 3 entries. For each entry $h \in H$, the algorithm calculates the accuracy (acc) of using $statlist$ to estimate the selectivity of g. The accuracy depends on the $errorfactor$ value in that entry, as well as the accuracy of each of the statistics in $statlist$ (line 7). We show how to calculate the accuracy in case of histograms later in this section. $MaxAcc$ represents the maximum accuracy that can be achieved if the selectivity of g estimated using the currently available statistics (lines 8-10). As a result, the metric s_1 can be calculated as $s_1 = 1 - MacAcc$ (line 12). The second metric, s_2, can be calculated as the ratio between the UDI and the table cardinality (line 13).

The total score of the table is computed as an aggregate function of the two metric values (e.g., a weighted sum). One way to use the aggregated score is to use a threshold of statistic importance; if the value of the total score exceeds a threshold s_{max}, statistics

Algorithm 3.3 Is a particular table important?

```
1:  function ShouldCollectStats(t, PG)
2:      for all g ∈ PG do
3:          H ← {h|h ∈ StatHistory; h.T = t, h.colgrp = columns(g)}
4:          MaxAcc ← 0
5:          for all h ∈ H do
6:              n ← |h.statlist|
7:              acc ← h.errorfactor * ∏_{i=1}^{n} accuracy(h.statlist[i], g)
8:              if acc > MaxAcc then
9:                  MaxAcc ← acc
10:             end if
11:         end for
12:         s_1 ← (1 − MaxAcc)
13:         s_2 ← min(UDI(t)/cardinality(t), 1)
14:         score ← f(s_1, s_2)
15:         if score ≥ s_max then
16:             return TRUE
17:         end if
18:     end for
19:     return FALSE
20: end function
```

must be collected on this table. As s_{max} approaches 1, no QSS are collected during compilation (same as traditional query processing). As s_{max} decreases, the system becomes more aggressive and tends to collect more statistics. If $s_{max} = 0$, all candidate QSS are collected. In our implemented prototype, the aggregate function is the average of the two scores. Section 3.3.3 elaborates on the effect of changing the value of s_{max} on the system performance.

The computation of the accuracy score acc (line 7 in Algorithm 3.3) depends on calculating the accuracy of the underlying statistics relative to the predicate group at hand. Since we store the QSS as histograms (cf. Section 3.2.2), we now show how to compute

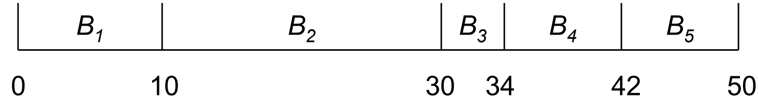


Figure 3.3: Sample histogram

the accuracy of a particular histogram in estimating the selectivity of a given predicate. The accuracy of a histogram with respect to a predicate (group) is a value in the range [0,1].

Consider a one-dimensional histogram on column a. A histogram has n buckets B_1, B_2, ..., B_n. Bucket B_i lies between the boundaries b_{i-1} and b_i. Now consider a predicate $a > value$ whose selectivity needs to be estimated from this histogram. The accuracy of the estimate depends on the following factors:

Distance from bucket boundaries If $value \approx b_i$ for some i, then estimating the selectivity is very accurate. As $value$ gets further from any boundary, the accuracy of the estimate decreases, since interpolation is needed within the bucket.

Bucket width The accuracy further decreases if $value$ lies within a wide bucket.

To calculate the accuracy for a one-dimensional histogram, we follow these steps:

1. Locate the bucket that contains $value$. Let this bucket be B_j with boundaries b_{j-1} and b_j.
2. Let $d_1 = value - b_{j-1}$ and $d_2 = b_j - value$
3. Let $u = \frac{min(d_1,d_2)}{max(d_1,d_2)} * \frac{b_j - b_{j-1}}{b_n - b_0}$
4. $accuracy = 1 - u$

Example 3.4. Consider the 5-bucket histogram in Figure 3.3. Given the steps outlined above, the accuracy of this histogram with respect to a predicate $a > 20$ is 0.6, while the accuracy of the same histogram with respect to a different predicate $a < 31$ is 0.97.

Algorithm 3.4 Is a statistic useful for other queries?

```
1: function ShouldMaterialize(g)
2:     if histogram exists on g then
3:         Return TRUE
4:     end if
5:     F ← Σ_{h∈StatHistory} h.count
6:     H ← {h|h ∈ StatHistory; columns(g) ∈ h.statlist}
7:     score ← Σ_{h∈H} (h.errorfactor * h.count/F)
8:     if score ≥ s_max then
9:         return TRUE
10:    else
11:        return FALSE
12:    end if
13: end function
```

For multi-dimensional histograms, we use a simple method where the overall accuracy can be computed as the product of the accuracy in each dimension.

2. Which Statistics to Materialize?

Once a table is sampled, the selectivities of all the candidate predicate groups given by the query analysis are computed and are used to optimize the query. However, we must decide which of these statistics to store in the QSS archive. We only need to store statistics that are potentially useful for future queries. JITS estimates the usefulness of materializing given statistics by monitoring how useful they were for previous queries. The usefulness score of a statistic depends on the number of times this particular statistic has been used in selectivity estimation, and the accuracy of the estimates it produces.

Algorithm 3.4 lists all history entries that have the statistic in question as one of their $statlist$ elements. For example, if the statistic in question is the predicate group *(a=5 AND b>10)*, and the history is as shown in Table 3.1, then H will have all entries whose $statlist$

contains the group (a, b), i.e. the first and fourth entries. The statistic is given a score that represents how beneficial it was for computing needed estimates. The algorithm uses a weighted average of $errorfactor$ to compute this score. If this score exceeds a certain threshold, it is considered useful, and is marked for materialization.

3.2.5 Updating the QSS Archive

Due to efficiency concerns, the statistics collected during query processing are not used to update the QSS archive as soon as they are obtained. Instead, the statistics are collected in a temporary buffer, and used in batches to update the QSS archive during periods of light load. Each entry in the buffer is in the form $(pg, count, t)$, where pg is a group of conjunctive predicates, $count$ is the number of rows that satisfy these predicates as collected by JITS, and t is a timestamp indicating collection time of this entry. Each predicate in pg is in the form *(exp relop C)*, where exp can be any expression involving table columns and $relop$ is a relational operator.

Updating the QSS histograms has to be performed such that each histogram is consistent with respect to the statistics stored in the buffer. The buffer is not cleared after the update process. An entry is only removed from the buffer when it has to be pruned to satisfy space constraints. The update process is based on the maximum entropy principle (cf. Section 2.4). We extended the technique in [49] to update the histograms by finding a distribution that satisfies the knowledge gained by the new statistics without assuming any further knowledge of the data, i.e., assuming uniformity unless more information is known. In the current prototype we limit exp to a column name. The value C has to be a constant value in order to be used for updating the histograms. For instance, for columns a and b, the predicate *($a < b + 10$)* cannot be used to update the histogram using maximum entropy[1]. The histograms in the QSS can represent data of most data types. Non-numeric data types, e.g. categorical and character types, can be represented as numerical values using a mapping function.

[1]Such predicates can still be stored in the buffer (or used to create statistical views as outlined in Chapter 4), and possibly reused for later queries.

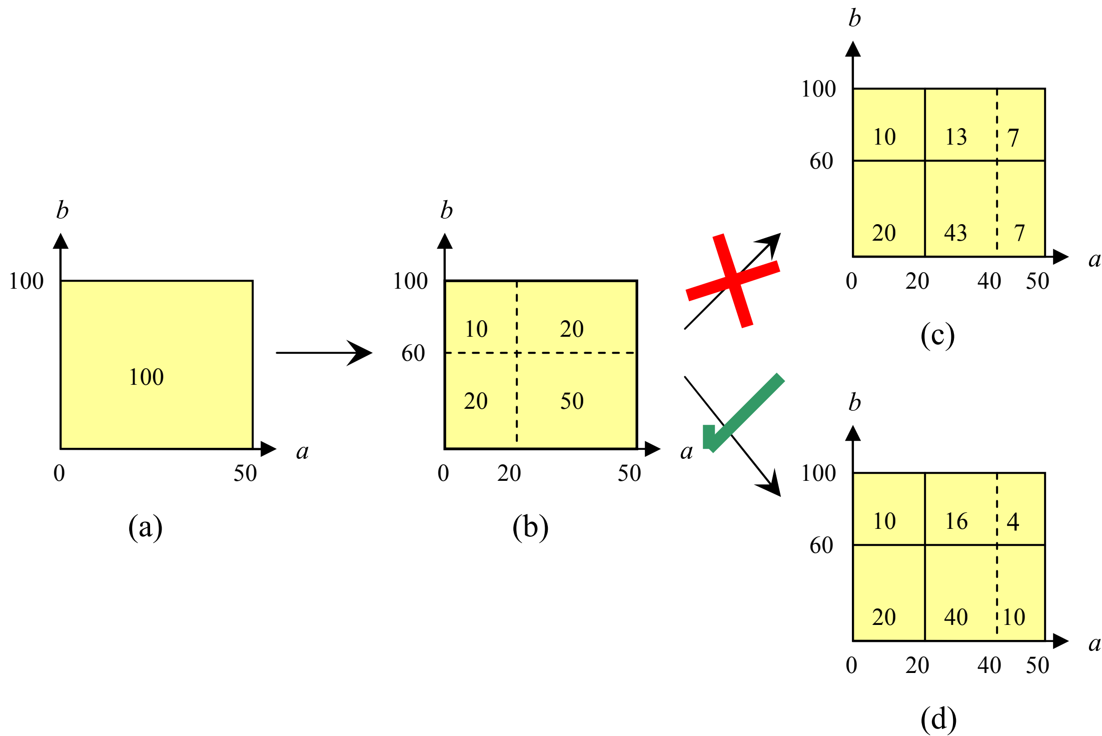


Figure 3.4: Histogram update

Example 3.5. Consider a 2-dimensional histogram on attributes a and b. The values of a range from 0 to 50 and the values of b range from 0 to 100. The total number of tuples is 100. Initially the histogram has just one bucket, as shown in Figure 3.4(a). Now consider a query with the predicates ($a > 20$ AND $b > 60$). After sampling, the system finds that 20 tuples satisfy this predicate group. This information is used directly to optimize the query. However, from the same sample we can determine the number of tuples that satisfy each of the 2 predicates individually (assume the number of tuples that satisfy $a > 20$ and $b > 60$ is 70 and 30, respectively). This new information is used to update the histogram as in Figure 3.4(b). Assume another query that has the predicate ($a > 40$), and assume there are 14 tuples satisfying this predicate. One possible way of incorporating this new information is the histogram shown in Figure 3.4(c), which is still consistent with all the

previously collected information. However, this histogram makes some assumptions about the distribution and the correlation between a and b, since the distribution of b for $a > 40$ is now different from the distribution of b for $20 < a < 40$.

Using the maximum entropy principle, and since no further information is known, we assume that a and b are independent, and we assume uniformity within the buckets in the last histogram. As a result, the newly inserted boundary splits the buckets as shown in Figure 3.4(d). It can be seen that the 20:50 ratio in Figure 3.4(b) is still maintained on both sides of the newly inserted boundary. Furthermore, the 56:14 (or 4:1) ratio of the tuples satisfying $20 < a < 40$ to the tuples satisfying $a > 40$ is also maintained whether b is above or below 60, which signifies the independence assumption.

As the system collects more statistics, storage space becomes an issue, especially since a single column can be involved in multiple histograms, each of which can be arbitrarily large. To avoid this problem, we keep a limit on the size of QSS to maintain. In case the dedicated space is full, and more statistics have to be materialized, we remove the buffer entries with the oldest timestamps, then rebuild any histograms that incorporated the deleted entries.

3.2.6 JITS Applicability

It is worth mentioning that JITS is more useful for complex, long-running queries such as those used in OLAP and Decision Support Systems. Such queries usually include a relatively large number of joined tables, aggregate functions, and predicates, which means more alternative plans to choose from. The long running time of these queries justifies spending time on statistics collection to guarantee the optimizer's access to recent accurate statistics, thus bringing down the total response time of the query. On the other hand, simple OLTP queries usually do not involve a large number of tables, and their running time is usually very short. For this reason, they might not benefit much from using the approach presented in this chapter. In fact, using such architecture can increase the time of query processing if all the queries are very simple. This is further illustrated in our experimental study.

Table 3.2: Table sizes

Table	No. of Tuples
CAR	1,430,798
OWNER	1,000,000
DEMOGRAPHICS	1,000,000
ACCIDENTS	4,289,980

3.3 Experimental Evaluation

We implemented the prototype within *DB2* [1]. The dataset that we used contains four relations: `CAR`, `OWNER`, `DEMOGRAPHICS`, and `ACCIDENTS`. Several primary-key-to-foreign-key relationships exist between the tables, as well as a number of correlations between attributes, such as `Make` and `Model`. Table 3.2 shows the number of tuples in each of the four tables.

The prototype uses the Query Graph Model (QGM) [32] to analyze the query structure. For statistics collection, the prototype invokes the RUNSTATS tool with the appropriate parameters. Based on earlier work [5, 34, 46], the best sample size sufficient to give accurate statistics of a database table is independent of the table size, and thus can be scaled to large tables. To collect specific predicate selectivities, we had to construct and invoke sampling queries on-the-fly.

As a future extension, techniques such as the work described in [52] can be employed to reduce the time used for sampling by making use of the existing catalog statistics.

3.3.1 JITS for a Single Query

To evaluate the benefit of our model, we issued a query given different scenarios. The query used for this experiment is:

```
SELECT o.name, driver, damage
FROM car c, accidents a, demographics d, owner o
WHERE d.ownerid = o.id
AND a.carid = c.id
AND c.ownerid = o.id
AND make = 'Toyota' AND model = 'Camry'
AND city = 'Ottawa' AND country = 'CA'
AND salary > 5000
```

This query was issued in 4 different scenarios:

1. No initial statistics (a) with JITS disabled, and (b) with JITS enabled
2. With Initial basic and distribution statistics on all tables (a) with JITS disabled, and (b) with JITS enabled

Table 3.3 shows the compilation, execution, and total times of the query under the different cases. The times shown are in seconds. Note that the total time is slightly larger than the sum of the compilation and execution times because it also includes the fetch time, which is the same in all cases.

Table 3.3: Compilation and execution times (in seconds)

Case #	Compilation	Execution	Total
1-a	0.098	138.756	138.855
1-b	11.864	101.681	113.547
2-a	0.073	104.912	104.986
2-b	3.698	101.323	105.023

In the first 2 cases, no statistics are known initially. When JITS is enabled (case 1-b), some overhead is encountered for collecting statistics. However, the execution time decreases significantly. The decrease in execution time is almost 27% and the overall gain is around 18% reduction in total query time. In the existence of all recent general statistics,

JITS might not outperform the traditional model for a single query. The reason is that the saving in the execution time can be outweighed by the JITS overhead. However, once we consider a sequence of queries in a workload, the overhead is amortized by reusing the statistics in the QSS archive. We show the workload effect in Section 3.3.2.

3.3.2 JITS for a Workload

This experiment demonstrates the performance of JITS as opposed to traditional query processing. We observed the performance of the system using a workload of 840 queries, including data updates to simulate a real-world operational database. Each query in the workload involves 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
```

The workload was executed in four settings:

1. JITS disabled, having no initial statistics
2. JITS disabled, having general (basic and distribution) statistics about all tables and columns
3. JITS disabled, having general (basic and distribution) statistics about all tables and columns in addition to workload statistics (i.e., all column groups that occur in all the queries)
4. JITS enabled, having no initial statistics

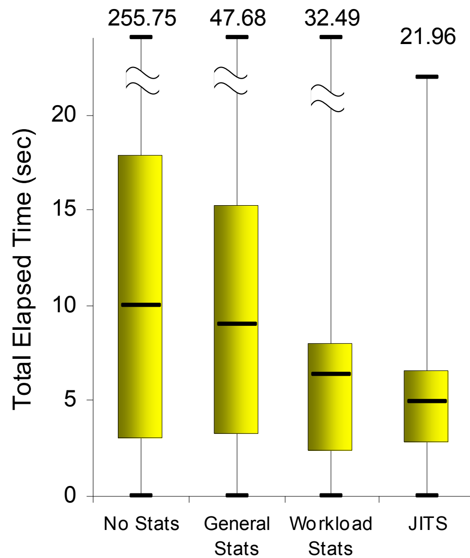
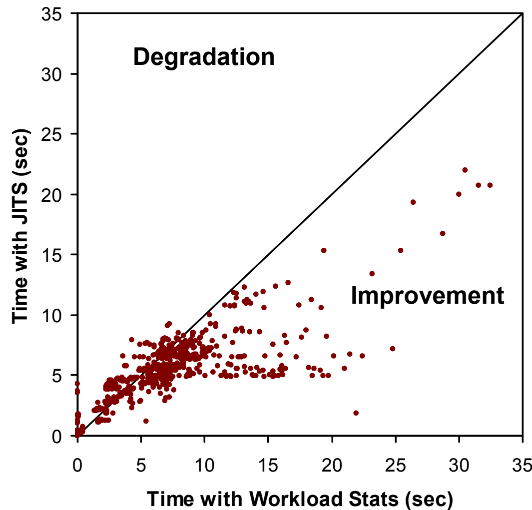


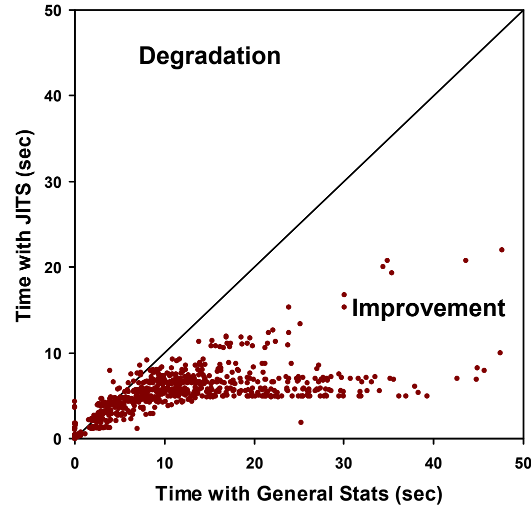
Figure 3.5: JITS benefit

In Setting (3), we assumed knowledge of the whole workload in advance. The set of "workload statistics" available in this setting are the QSS that would be recommended by JITS for all the queries in the workload. We had all these statistics collected and available to the optimizer before running the workload. Figure 3.5 shows a box plot (a graph depicting the smallest observation, lower quartile, median, upper quartile and largest observation) of the elapsed time of the workload queries in the four settings. Having general statistics only results in a significant benefit compared to having no statistics at all. However, if workload information is available, it can be analyzed and all the needed statistics can be collected beforehand, which improves the overall performance. However, due to data updates, these statistics soon become stale, and the estimation error increases.

With JITS enabled, the system samples the data to get the actual selectivities of the predicates in the query. The benefit of having these very accurate values outweighs the sampling overhead. In addition, the data updates have no effect on the accuracy of the collected statistics, since the system detects the staleness of these statistics, and recollects them when needed, which justifies the performance gain of setting (4) over setting (3).



(a) Workload stats vs. JITS



(b) General stats vs. JITS

Figure 3.6: Individual query performance

Figure 3.6(a) shows a scatter chart of the elapsed times of individual queries when JITS is enabled (with no prior statistics) versus when JITS is disabled (having workload statistics to start with). Some of the queries suffer from the overhead of collecting statistics when JITS is enabled. We observed that most of these queries are in or near the beginning of the workload, where the collected workload statistics are still valid. As the data gets updated, the workload statistics become stale, and the benefits of JITS become evident. In addition, queries that have very short execution times (bottom left corner of the chart) often do not benefit from JITS, since the overhead in optimization time is not justified.

In the majority of the systems, where prior workload knowledge is not available, only general statistics can be collected initially. Figure 3.6(b) depicts JITS versus having general statistics. Almost all of the queries have a significant improvement, while only a few ones lie in the degradation region.

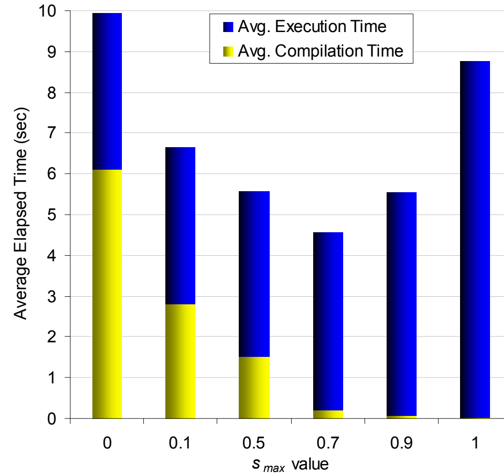


Figure 3.7: Sensitivity analysis threshold

3.3.3 Tuning the Sensitivity Analysis

As mentioned in Section 3.2, the sensitivity analysis module determines whether or not to collect certain statistics. Each statistic is given an overall score based on individual scoring factors. A statistic is to be collected if its overall score exceeds a certain threshold s_{max}.

We used the same workload used in Section 3.3.2. Figure 3.7 shows the average elapsed time per query for $s_{max} = 0, 0.1, 0.5, 0.7, 0.9$, and 1. At $s_{max} = 0$, all possible statistics are always collected, i.e., there is no actual sensitivity analysis. This explains the very large compilation time. The compilation time decreases as s_{max} increases since fewer statistics are collected. At $s_{max} = 1$, no statistics are ever collected (similar to traditional query processing). Note that if there is no sensitivity analysis ($s_{max} = 0$), our system performs worse than traditional query processing ($s_{max} = 1$) because of the added overhead. Increasing s_{max} from 0 to 0.5 decreases the average compilation time significantly while the average execution time is not affected, which means that there has been useless statistics collection at the lower values of s_{max}. At $s_{max} = 0.7$, there is an increase in the average execution time, outweighed by the decrease in the average compilation time. This means that setting

$s_{max} = 0.7$ might be a better choice if we have a workload. However, $s_{max} = 0.5$ would be better for a single query (where the system collects the minimum amount of statistics to achieve the least possible execution time).

3.4 Conclusions

This chapter presents an efficient approach to proactively determine, collect, and materialize Just-in-Time Statistics (JITS) during query processing. In contrast to similar research efforts, the work presented here: (1) employs a lightweight sensitivity analysis based on the query structure, the existing statistics and the data activity to identify the crucial statistics; and (2) materializes and incrementally updates the collected partial statistics for future reuse. Since the statistics cover partial (possibly overlapping) regions of the data space, our technique integrates these partial statistics in a reusable form by maintaining maximum-entropy-based structures. Our proposed framework can easily be extended to employ more sophisticated sensitivity analysis techniques. We demonstrated our approach and evaluated its benefits through an extensive experimental study.

Collecting Just-in-Time Statistics guarantees that the optimizer has access to statistics that are recent and truly representative of the underlying data, including any possible correlations. The presence of these statistics minimizes the effect of the optimizer's uniformity and independence assumptions, and significantly reduces cost estimation error. Using this approach is more useful for complex ad-hoc long-running queries such as the ones used in OLAP and Decision Support Systems since investing some time to collect statistics significantly reduces the total query response time. Even in the case of known workloads (as opposed to ad-hoc queries), data updates can cause stored statistics to be stale quickly. JITS detects such cases, and updates the stored statistics to ensure they reflect the underlying data.

Lessons Learned. Based on the study and the experiments presented in this chapter, we make the following high level observations:

- For short-running queries, if general statistics are available and recent, enabling JITS may in fact hurt the performance, since the overhead associated with JITS does not result in significant savings in execution time.
- Query-specific statistics provide better estimate accuracy than general statistics, even if the QSS get out of date. This was evident when having workload statistics caused a significant performance improvement over having just general statistics.
- JITS is useful for single ad-hoc queries as well as workloads that include data updates.

Chapter 4

Recommending Statistical Views

In Chapter 3, we presented a technique to quickly determine and collect the necessary statistics to optimize a given single query. However, if a whole read-only workload is known in advance, using JITS for every query is not the best approach, since it means lots of repeated work, and potentially sampling multiple times to collect very similar information. For example, one query requires the selectivity of predicate $(x < 100)$ while another query requires the selectivity of predicate $(x < 150)$. For such workloads, it might be better to analyze the whole workload at once and exploit similarities between queries. Since such workload is known in advance, it is possible to perform this analysis offline and not as part of query processing. In addition, instead of only recommending predicate groups, it would be more beneficial to create and maintain more complex specific statistics that can capture correlations between multiple tables as well as non-uniform data distributions, and that can cause the optimizer to choose different execution plans as a result of more accurate cost estimates.

In this chapter, we describe the work we proposed in [23] to automatically recommend statistical views (statviews) for a given SQL workload. We define the problem of recommending statviews, and outline the architecture of the *StatAdvisor* framework in Section 4.1. We describe our novel plan-based candidate enumeration technique in Section 4.2. Section 4.3 introduces our benefit metric, which is later used to select the final

recommendations. The dependency of the *StatAdvisor* on the database engine is discussed in Section 4.4. We demonstrate our experimental results in Section 4.5, and summarize this chapter in Section 4.6.

4.1 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}.*

4.1.1 Key Insights

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

Observation 4.1. *Statviews (or statistics in general) have a direct effect on execution cost only when they cause the optimizer to 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 (hopefully 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 different one.

Example 4.1. Consider a query Q. When Q is optimized given only the base table statistics in the catalog, the optimizer chooses plan P, and estimates the execution cost to be E_0. Now suppose statview v_1 is created and statistics on it are collected. Optimizing Q given the new statview yields the same plan P but with estimated cost $E_1 < E_0$. Now consider a different scenario where a different statview v_2 is created. Given the statistics on v_2, optimizing Q yields a different plan P' with estimated cost $E_2 > E_0$. In the case of v_1, even though the

estimated cost E_1 is less than the estimated cost E_0 obtained without v_1, it is evident that v_1 has no effect on the chosen execution plan, and hence the *actual* cost of executing Q is the same with and without v_1. The difference in the estimated costs is merely because these estimates are computed from different sets of statistics. In the case of v_2, the optimizer choose a different plan, which will have a different execution cost. Thus v_2 does indeed have an effect on the execution cost of Q.

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, and not merely compare the cost estimates obtained with and without the statviews. The use of plan change as a measure of statistics relevance has been previously used in [18] 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 4.2. *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) to 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 [13]). 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 4.1). We explain in Section 4.3 how the benefit of a statview is estimated in *StatAdvisor*.

Observation 4.3. *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.

Example 4.2. Consider the query plan at the top left corner of Figure 4.1. 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 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. [13, 18]) 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 4.3).

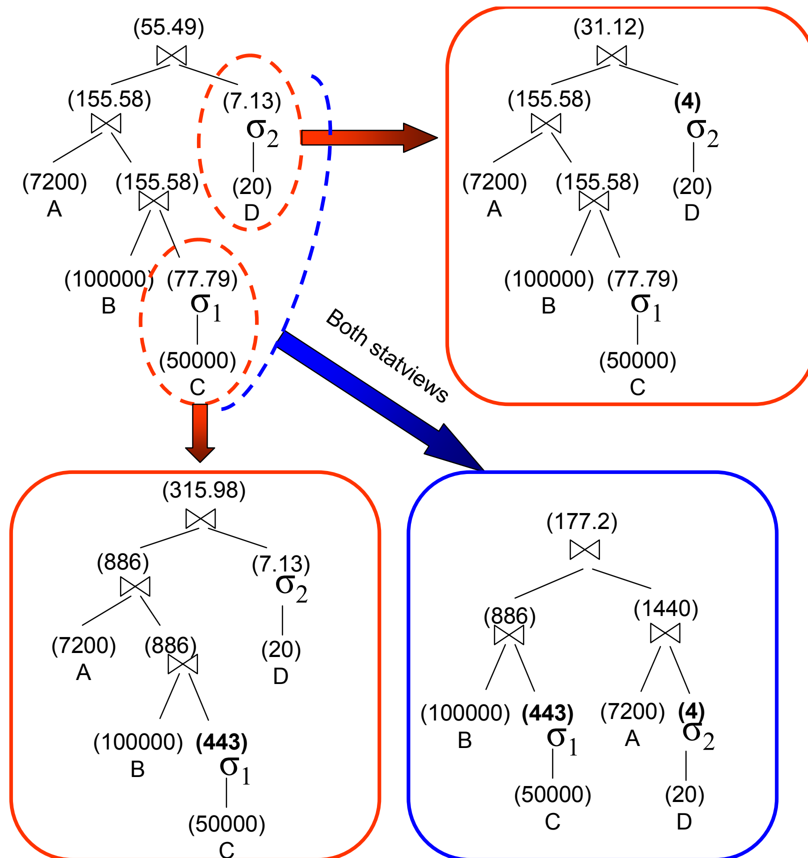


Figure 4.1: Effect of statview-groups

Observation 4.4. *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:

Definition 4.1. *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 4.2. *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, there are two reasons why the optimizer might pick a bad (expensive) execution plan: Either (a) the cost of the selected (bad) plan is underestimated, making it seem cheaper than it actually is, and hence more attractive for the optimizer, or (b) the cost of an unselected (good) plan is overestimated, making it less attractive for the optimizer. Either of these reasons (or both) can lead to the optimizer choosing a sub-optimal plan. 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 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 4.2.1. 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 and the data (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 (page 2)
- 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 [13]. 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 [10].

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 4.2.1.

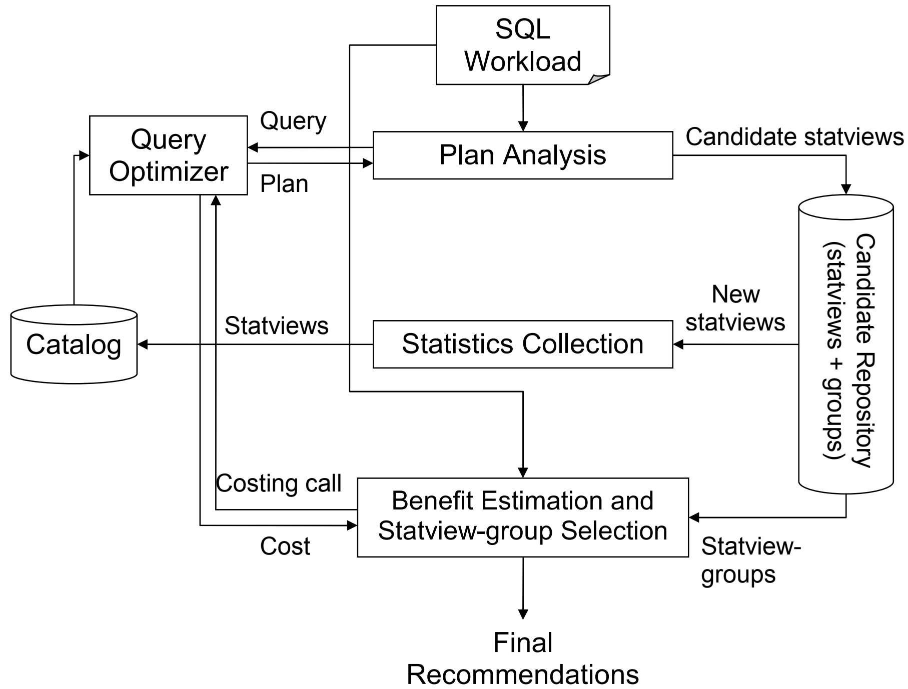


Figure 4.2: *StatAdvisor* architecture

4.1.2 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 [8, 51, 56]. However, the computation of the benefit estimates takes into account the special characteristics of statviews and their effect on query performance.

Figure 4.2 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 4.2). The candidate statviews are stored in

Algorithm 4.1 StatAdvisor

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

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 4.3).

Algorithm 4.1 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 2). Subsequently, the algorithm performs a number of iterations while V is not empty (lines 3-6). 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.

4.2 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 4.2.1, then we extend it to process the whole workload in Section 4.2.2.

4.2.1 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 4.4, our objective is to ensure that the cost of the chosen plan 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 however, collecting all statviews is unnecessary and extremely expensive. Hence, we define *important statviews* as follows:

Definition 4.3. 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.

Example 4.3. Figure 4.3 depicts a query execution plan. The marked expressions, referencing one or two base tables, constitute the important statviews for this plan.

In our experiments (Section 4.5.2), 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

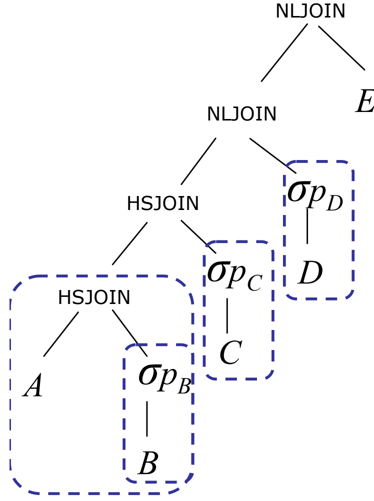


Figure 4.3: Important statviews

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 4.1). 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.

Example 4.4. Consider the plan P_0 in Figure 4.4(a). According to Definition 4.3, the important statviews for this plan are:

v_1**:** $\sigma_{p_B}(B)$

v_2**:** $\sigma_{p_C}(C)$

v_3**:** $\sigma_{p_D}(D)$

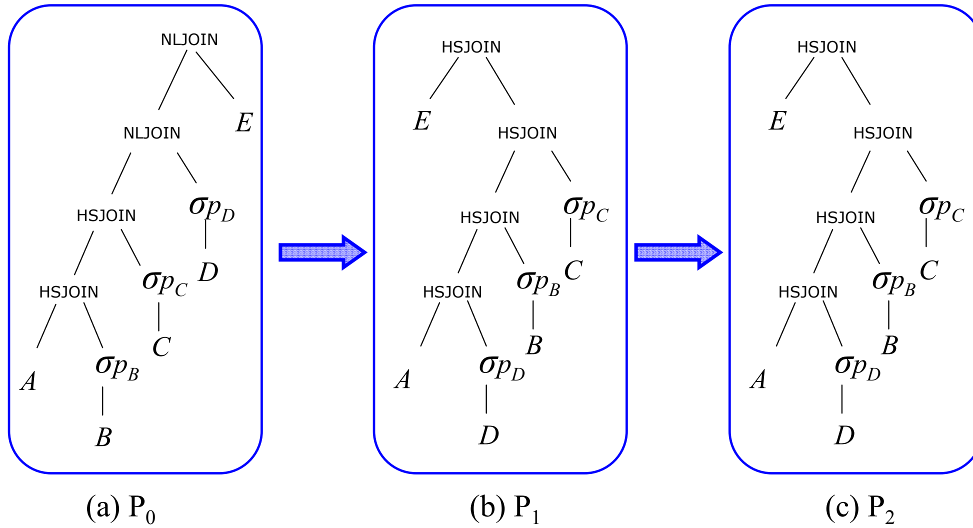


Figure 4.4: Candidate enumeration

v_4**:** $A \bowtie \sigma_{p_B}(B)$

After collecting these statviews in P_0 and re-optimizing, the obtained plan is P_1 (Figure 4.4(b)). The important statviews for this plan are:

v_3**:** $\sigma_{p_D}(D)$

v_1**:** $\sigma_{p_B}(B)$

v_2**:** $\sigma_{p_C}(C)$

v_5**:** $A \bowtie \sigma_{p_D}(D)$

The first 3 statviews have already appeared in the previous plan and have already been collected. Only the last statview v_5 is new. After collecting this statview and re-optimizing, 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 4.3, 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 the costs of 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 (since 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 4.4, 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 4.4).

4.2.2 Candidate Enumeration for a Workload

The naïve approach to obtain the candidates for the whole workload is to repeat the technique described in Section 4.2.1 for each query separately. 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 [7], thus causing the respective sample or join synopsis to be read from disk only once. Algorithm 4.2 gives our approach to produce the candidates with respect to the whole workload W of n queries. Algorithm 4.2 provides the details of lines 2-6 in Algorithm 4.1. 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 3-5). 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 6-21. 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 obtained (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), there is no need for it to be 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 a plan that has been seen before, or a different plan whose important statviews have already been collected, processing is stopped for that query (as in Section 4.2.1). Only the remaining queries are entered into the next iteration. Processing stops when no queries are left.

Suppose we have an oracle that can tell us all the statviews that we need for each query. If that was the case, we would group all statviews that have the same set of tables (from all queries), and collect them simultaneously from the corresponding sample, thus only touching that sample once. However, since such oracle does not exist, we need to perform multiple iterations, potentially collecting information from the same sample more

Algorithm 4.2 Candidate enumeration

```
1: function CANDIDATEENUMERATION(W)
2:     Assume that W = {Q_1, Q_2, ...Q_n}
3:     Let V_i ← φ and finished_i ← false for i = 1 → n
4:     for all Q_i ∈ W : P_{i,0} ← Optimize(Q_i), VT_i ← FindImportantStatviews(P_{i,0})
5:     k ← 1
6:     while ∃i s.t. finished_i = false do
7:         V = {⋃_i VT_i | finished_i = false}
8:         {U_1, U_2, ..., U_m} ← Partition(V)
9:         Get a sample (or join synopsis), and collect all statviews in U_j for j = 1 → m
10:        for all Q_i ∈ W s.t. finished_i = false do
11:            P_{i,k} ← Optimize(Q_i)
12:            if P_{i,k} = P_{i,l} for some l < k then
13:                finished_i ← true
14:            else (new plan)
15:                V_i ← V_i ∪ VT_i
16:                VT_i ← FindImportantStatviews(P_{i,k}) − V_i
17:                if (VT_i = φ) then finished_i ← true
18:            end if
19:        end for
20:        k ← k + 1
21:    end while
22:    return V_1, V_2, ...V_n
23: end function
```

than once. Our experiments show that most queries require 1-3 iterations before termination. The convergence of the algorithm is addressed in detail in our experimental evaluation (Section 4.5.2). 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 [18]; 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.

4.3 Benefit Estimation and Statview Selection

As mentioned in Section 4.1, 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 4.3.1, we define the benefit of a statview-group to a particular query, and in Section 4.3.2, we extend that definition to the whole workload. Section 4.3.3 presents our selection algorithm that exploits these benefit estimates.

4.3.1 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:

$$B(V, Q) = ActCost(Q, P_0) - ActCost(Q, P_V) \tag{4.1}$$

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 4.5. 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:

$$B'(V, Q) = EstCost(P_0, V) - EstCost(P_V, V) \tag{4.2}$$

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 $B'(V, Q) = 0$.

4.3.2 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:

$$B(V, W) = \sum_{Q \in W_V} B'(V, Q) \tag{4.3}$$

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 $B(V)$. Note that this formula may actually underestimate the benefit of some statview groups. For example, if statview group V_1 is recommended by query Q_1, and statview group $V_2 \subset V_1$ is recommended by query Q_2, then based on our formula, the benefit of V_1 only considers its effect on Q_1, even though collecting V_1 actually benefits both queries. This can be easily incorporated in the formula by changing it to consider all queries that recommended subsets of V. However, our experimental evaluation was performed with the above benefit calculation.

4.3.3 Statview-Group Selection

At this point, we have a set of statview-groups $G = \{V_1, ..., V_n\}$ where n is less than or 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.

Algorithm 4.3 Statview group selection

```
1:  function STATVIEWGROUPSELECTION(G, c_max)
2:      c ← 0, R ← φ
3:      while (|G| > 0 ∧ c < c_max) do
4:          V_best ← null
5:          B_best ← 0
6:          for all V ∈ G do
7:              if (B(V) > B_best ∧ C(V, R) ≤ c_max − c) then
8:                  V_best ← V
9:                  B_best ← B(V)
10:             end if
11:         end for
12:         if (V_best ≠ null) then
13:             c ← c + C(V_best, R)
14:             G ← G − V_best
15:             R ← R ∪ V_best
16:         else
17:             Break
18:         end if
19:     end while
20:     return R
21: end function
```

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.

In our implementation, we use the greedy solution given in Algorithm 4.3. 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 less than or equal to the number of queries in the workload).

Note that the cost constraint c_{max} can be viewed as a *budget* allocated for statviews. In many cases, it is possible not to set any particular constraint, in which case, the system will recommend all statview groups with a positive benefit.

4.4 Dependency on Database Engine

The implementation details and some algorithms of *StatAdvisor* depend 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 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. 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.)

As seen in Section 2.3.4, statview matching is different from materialized view matching. 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. We discuss statview matching in more detail and outline our statview matching technique in Chapter 5.

4.5 Experimental Evaluation

In this section, we present experimental results of an implementation of the *StatAdvisor* in *DB2* [1].

4.5.1 Setup

Data: We carried out our experiments on two different data sets. The first data set, DS_1, is a TPC-DS [4] 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

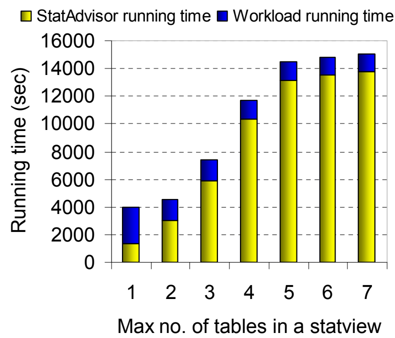
do not contain subqueries[1]. However, the selected queries include various constructs, e.g. arithmetic expressions, functions, equi-joins, range joins, equality and range filtering predicates, conjuncts and disjuncts. Each query consists of three to seven joined tables. The second workload, W_2, corresponds to the second data set (DS_2), and contains 100 synthetically generated SPJG queries. Each query joins one to five 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
```

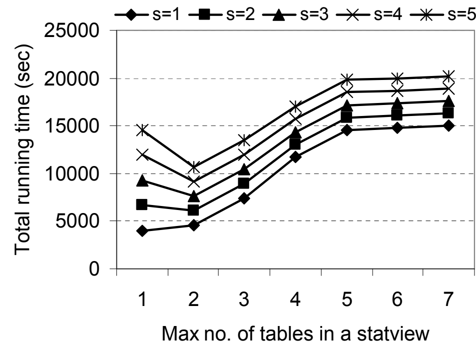
4.5.2 Candidate Enumeration

As mentioned in Section 4.2.1, 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 4.5(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 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 outweighed 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 $t = 2$. This is mainly a result of the schema of the data. Since both databases

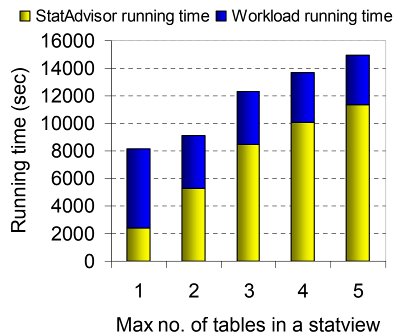
[1] At the time of performing these experiments, *DB2* does not match statviews with queries that include subqueries. However, *StatAdvisor* can generally recommend statviews that correspond on any query expression, including subqueries.



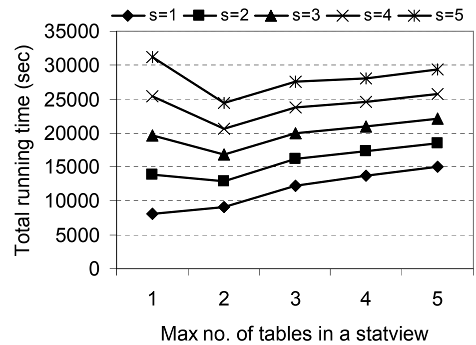
(a)



(b)



(c)

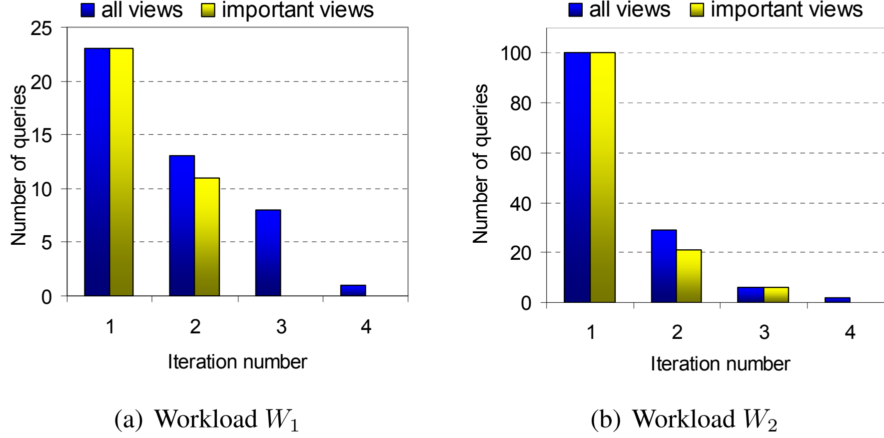


(d)

Figure 4.5: Important statviews

were based on a star-schema, having views over 2 tables where one of these table is the fact table is sufficient to capture most data correlations.

Generally, the total running time = (*StatAdvisor* running time + s * workload execution time), where s is the number of executions of the workload. Figure 4.5(b) depicts the total running time against the maximum number of tables in a statview (t), for different values of s. The best performance for most values of s occurs at $t = 2$. Figures 4.5(c) and 4.5(d) depict the corresponding results for the second workload W_2, in which the same effect can be seen. Based on these results, we decided to limit the important statviews to those with one or two joined tables, along with their selection predicates, since including more



(a) Workload W_1 (b) Workload W_2

Figure 4.6: Convergence of candidate enumeration

statviews increases the overhead without introducing significant benefit.

Figure 4.6 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 4.3) are collected. The x-axis represents the various iterations, and the y-axis represents the number of queries being processed in each iteration. Figures 4.6(a) and 4.6(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 is as follows:

Consider a query Q. At the kth 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, 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.

4.5.3 Overall Workload Performance

The test workloads are executed in the following settings:

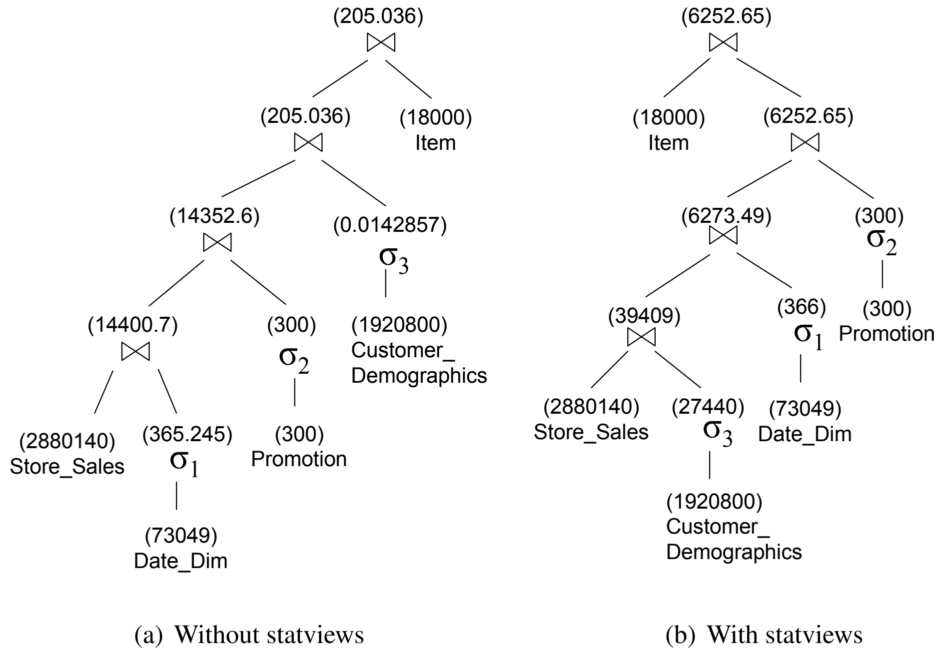
1. Statistics are available on all base tables and their attributes, including table cardinalities, number of distinct values in each column, etc. This represents the common case in most database systems.

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. We did not provide a cost contraint, so the system recommended all statview groups that had a positive benefit. 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.



(a) Without statviews (b) With statviews

Figure 4.7: Execution plans for TPC-DS query 7

Example 4.6. Consider the following query:

```
SELECT i_item_id, ss_quantity, ss_list_price, ss_coupon_amt, ss_sales_price
FROM Store_Sales, Customer_Demographics, Date_Dim, Item, Promotion
WHERE ss_sold_date_sk = d_date_sk AND ss_item_sk = i_item_sk
AND ss_cdemo_sk = cd_demo_sk AND ss_promo_sk = p_promo_sk
AND cd_gender = 'M'
AND cd_marital_status = 'S'
AND cd_education_status = 'College'
AND (p_channel_email = 'N' OR p_channel_event = 'N')
AND d_year = 2000
```

This is an example of a query that benefited from the recommended statviews in W_1. The query includes three filtering predicates on the *Customer_Demographics* table, one predicate on the *Date_Dim* table, and two disjunctive predicates on the *Promotion* table.

Figure 4.7(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: `SELECT d_date_sk FROM Date_Dim WHERE d_year = 2000`

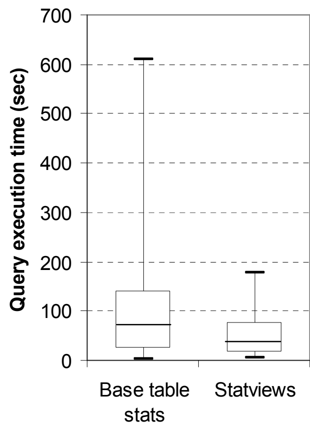
v_2: `SELECT p_promo_sk FROM Promotion WHERE p_channel_email = 'N' OR p_channel_event = 'N'`

v_3: `SELECT cd_demo_sk FROM Customer_Demographics WHERE cd_gender = 'M' AND cd_marital_status = 'S' AND cd_education_status = 'College'`

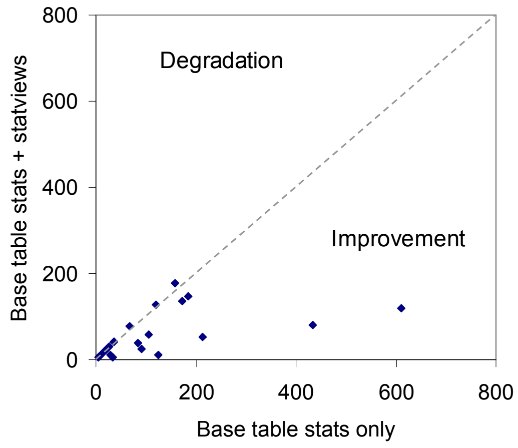
v_4: `SELECT ss_item_sk, ss_cdemo_sk, ss_promo_sk FROM Store_Sales, Date_Dim WHERE ss_sold_date_sk = d_date_sk AND d_year = 2000`

Figure 4.7(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).

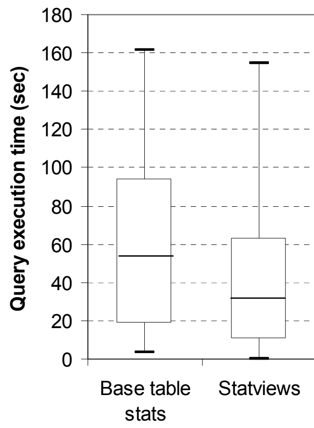
Figure 4.8(a) is a box plot (a graph depicting the smallest observation, lower quartile, median, upper quartile and largest observation) of the execution time of the queries of W_1 in the two settings. Figure 4.8(b) shows a scatter plot of the elapsed times of the individual queries of W_1, where the 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 4.8(c) and 4.8(d) represent



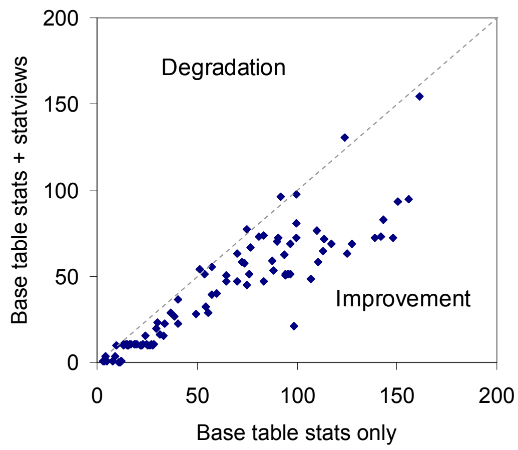
(a)



(b)



(c)



(d)

Figure 4.8: Workload performance

Table 4.1: *SITadvisor* vs. *StatAdvisor*

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

the corresponding results for W_2. As can be seen, the introduction of statviews significantly improves the performance of the system.

4.5.4 Comparison with Previous Work

For this experiment, we implemented the statistics selection algorithm given in [13]. 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 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 4.1 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 4.9 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-

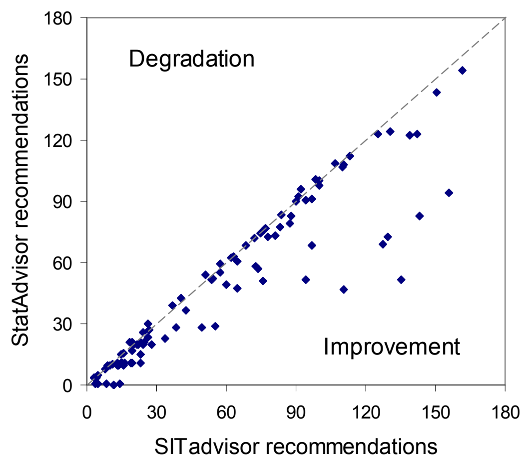


Figure 4.9: *SITadvisor* vs. *StatAdvisor*

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.

4.6 Conclusions

This chapter outlines *StatAdvisor*, our framework for automatic recommendation of 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. 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. We presented the basic concepts and insights on which our approach is based. We also outlined the architecture, and key features of *StatAdvisor*, and demonstrated its validity and benefits through an extensive experimental study using a

prototype that we built in the *IBM DB2* database system.

Lessons Learned. Based on the study and the experiments presented in this chapter, we make the following high level observations:

- Collecting statistics on statviews gives the optimizer more accurate cardinality estimates at various points in the query plan, thus allowing the optimizer to estimate 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.
- For data warehousing schemas, statviews that reference one or two tables are sufficient to capture the correlations between the table attributes.
- Taking into account the dependencies between statviews and analyzing the plans produced by the optimizer produces better statview recommendations.

Chapter 5

Exploiting Statviews for Selectivity Estimation

In Chapter 4 we presented our approach to recommend statistical views or statviews for a given SQL workload. As outlined in the introduction (Section 1.2), the presence of statviews introduces the problem of having multiple, potentially non-equivalent ways of estimating the selectivity of a given set of predicates. Choosing one way over another arbitrarily biases the optimizer toward choosing one plan over the other. In addition, if different estimates are used every time it is required, then different plans will be costed inconsistently, leading to incorrect comparisons and unreliable plan choices.

In this chapter, we discuss our implementation of two methods of exploiting statviews in query optimization based on the work in [14] and [42]. The first method analyzes all possible ways of computing a selectivity estimate, and uses the one that has the minimum estimated error. The second method relies on the principle of maximum entropy to make use of all the available information while computing the estimate. We adapted the two methods to work in the context of statviews, and implemented them inside the *PostgreSQL* [3] query optimizer.

First, we define the problem and outline the notations we use in Section 5.1. We then proceed to explain the theory behind the two approaches we used, namely the conditional

selectivity approach [14] in Section 5.2, and the maximum entropy approach [42] in Section 5.3. We discuss the view matching conditions in Section 5.4, and the integration with the optimizer in Section 5.5. We demonstrate our experimental results in Section 5.6, and summarize this chapter in Section 5.7.

5.1 Notations and Problem Definition

In this section, we start by establishing formal definitions for selectivity, database statistics, and statviews, then we present the problem statement. For the set of tables $\mathcal{R} = \{R_1, ..., R_n\}$, let $\mathcal{R}^\times$ refer to the Cartesian product $R_1 \times ... \times R_n$. Also, for a set of predicates P, let $tables(P)$ denote the set of tables referenced by the predicates in P, and $attr(P)$ denote the set of attributes referenced in those predicates.

Definition 5.1. (SELECTIVITY) For a set of tables $\mathcal{R}$ and a set of predicates P over $\mathcal{R}^\times$, the *selectivity* $Sel_{\mathcal{R}}(P)$ denotes the fraction of tuples in $\mathcal{R}^\times$ that simultaneously satisfy all the predicates in P. In general, each predicate in P can also be a disjunction of predicates.

Base table statistics are usually available on the attribute level. If statistics are available for attribute a_1 of table R_1, these statistics can be used to estimate the selectivity $Sel_{R_1}(P)$ if $tables(P) = \{R_1\}$ and $attr(P) = \{a_1\}$.

Definition 5.2. (STATVIEWS) Let $\mathcal{R} = \{R_1, ..., R_n\}$ be a set of tables, $P = \{p_1, ..., p_m\}$ be a set of predicates over $\mathcal{R}^\times$, and $A = \{a_1, ..., a_k\}$ be a subset of the attributes in $\mathcal{R}^\times$. A statview v can be expressed as: $v = SV_{\mathcal{R}}(A|P)$ if v is defined as the query $\pi_{a_1,...,a_k}(\sigma_{p_1 \wedge ... \wedge p_m}(R_1 \times ... \times R_n))$.

Example 5.1. Let v be a statview defined by the query `SELECT a, b FROM R WHERE c < 100 AND d=2` . Using our notation, we can say that $v = SV_R(a, b|c < 100 \wedge d = 2)$.

To simplify the presentation throughout this chapter, we use "P, Q" to denote "$P \cup Q$" and "p, Q" to denote "$\{p\} \cup Q$", where p is a predicate and P and Q are sets of predicates.

Problem Statement

Given an SPJ query expression $\sigma_{p_1 \wedge ... \wedge p_m}(R_1 \times ... \times R_n)$, the required task is to estimate $Sel_{\mathcal{R}}(p_1, ..., p_m)$, the fraction of tuples in $R_1 \times ... \times R_n$ that simultaneously satisfy all the predicates $p_1, ..., p_m$. The selectivity estimate should be computed given multiple base table statistics and statviews that can partially contribute to that estimate.

This problem definition applies to any SPJ query expression. This could be a whole query, or a sub-expression of a query. Each sub-expression would have a subset of the predicates and the relations of the query, and its required selectivity would be based on those subsets.

In our work, we implemented two methods to estimate the selectivity of a group of conjunctive predicates given statviews that partially covers that group. The two methods are based on the work in [14] and [42], and are discussed in Sections 5.2 and 5.3 respectively.

5.2 Estimation using Conditional Selectivity

The work in [14] introduces the concept of *conditional selectivity*, which allows expressing the selectivity of a given SPJ query in many different but equivalent ways.

Definition 5.3. Given a set of tables $\mathcal{R}$, and two sets of predicates $P = \{p_1, ..., p_j\}$ and $Q = \{q_1, ..., q_k\}$ over $\mathcal{R}^\times$, the *conditional selectivity* $Sel_{\mathcal{R}}(P|Q)$ is defined as the fraction of tuples in $\sigma_{q_1 \wedge ... \wedge q_k}(\mathcal{R}^\times)$ that simultaneously satisfy all predicates in P. In other words:

$$Sel_{\mathcal{R}}(P|Q) = \frac{|\sigma_{p_1 \wedge ... \wedge p_j}(\sigma_{q_1 \wedge ... \wedge q_k}(\mathcal{R}^\times))|}{|\sigma_{q_1 \wedge ... \wedge q_k}(\mathcal{R}^\times)|} \tag{5.1}$$

If $Q = \phi$, then $Sel_{\mathcal{R}}(P|Q)$ falls back to the original definition of selectivity $Sel_{\mathcal{R}}(P)$.

The work in [14] refers to SITs (statistics on intermediate tables), each of which is a histogram. To be consistent with our own terminology, our more general statview $v =$

$SV_{\mathcal{R}}(A|P)$ can be viewed as a group of SITs, one for every attribute in A. A given SIT (or histogram) H_a on attribute $a \in A$ can be used to estimate the selectivity $Sel_{\mathcal{R}}(q|P')$ for any predicate q where $attr(q) = a$ and $P' \supseteq P$. Information about the statview itself, e.g. $|v|$ is not used in this approach.

Example 5.2. Consider a statview $v = SV_{R_1}(a, b|c < 100)$. This statview provides the two histograms H_a and H_b. The histogram H_a, for instance, can be used to estimate the selectivity $Sel_{R_1}(a > 5|c < 100)$. The same histogram can also be used to estimate the selectivity $Sel_{R_1}(a > 5|c < 100 \wedge d = 2)$ by assuming independence between the predicates $a > 5$ and $d = 2$.

To achieve the objective described in Section 5.1, a key property of conditional selectivity is *atomic decomposition*, which uses the notion of conditional probability to represent a selectivity value as the product of two conditional selectivity values.

Property 5.1. (ATOMIC DECOMPOSITION) Given a set of tables $\mathcal{R}$ and sets of predicates P and Q:

$$Sel_{\mathcal{R}}(P, Q) = Sel_{\mathcal{R}}(P|Q) \cdot Sel_{\mathcal{R}}(Q) \tag{5.2}$$

This property holds for arbitrary sets of predicates and tables, without relying on any simplifying assumptions. An atomic decomposition divides the problem of estimating $Sel_{\mathcal{R}}(P, Q)$ into two sub-problems (estimating $Sel_{\mathcal{R}}(P|Q)$ and $Sel_{\mathcal{R}}(Q)$). The decomposition property is essential to develop a framework for exploiting statviews. Statviews can be used to estimate the first factor $Sel_{\mathcal{R}}(P|Q)$, as in Example 5.2. In turn, if Q consists of a single predicate we can use standard estimation techniques to estimate the second factor $Sel_{\mathcal{R}}(Q)$, or otherwise recursively apply another atomic decomposition to $Sel_{\mathcal{R}}(Q)$.

By repeatedly applying atomic decompositions, a given selectivity estimate may result in a very large number of alternative expressions. These expressions can be referred to as *decompositions* of the original selectivity value. A decomposition of a given selectivity value is then an expression of the form $S_1 \cdot ... \cdot S_k$, where each $S_i = Sel_{\mathcal{R}_i}(P_i|Q_i)$ and $Q_k = \phi$.

Example 5.3. Consider the selectivity estimate $Sel_{R_1}(a > 5 \wedge c < 100 \wedge d = 2)$. The following are some of the possible decompositions of this estimate:

- $Sel_{R_1}(a > 5 | c < 100 \wedge d = 2) \cdot Sel_{R_1}(c < 100 | d = 2) \cdot Sel_{R_1}(d = 2)$
- $Sel_{R_1}(a > 5 | c < 100 \wedge d = 2) \cdot Sel_{R_1}(d = 2 | c < 100) \cdot Sel_{R_1}(c < 100)$
- $Sel_{R_1}(d = 2 | a > 5 \wedge c < 100) \cdot Sel_{R_1}(a > 5 | c < 100) \cdot Sel_{R_1}(c < 100)$

If each factor $Sel_{\mathcal{R}_i}(P_i|Q_i)$ is calculated accurately, then every possible decomposition of $Sel_{\mathcal{R}}(P)$ evaluates to the same value. However, in reality, only a small number of statviews are available, besides the base table statistics. It follows that, depending on the available statistics, the optimizer has to perform some approximations, interpolations, and employ some independence and/or uniformity assumptions, causing some decompositions to be more accurate than others.

Suppose that each decomposition of a selectivity value $Sel_{\mathcal{R}}(P)$ is assigned a measure of how accurately the decomposition can be estimated using the current set of available statistics. Then the problem statement in Section 5.1 can be redefined as the following optimization problem: it is required to obtain the "most accurate" decomposition of $Sel_{\mathcal{R}}(P)$ for the given set of available statistics (see Section 5.2.2 for the definition of accuracy).

In principle, this problem can be approached as follows:

1. Exhaustively enumerate all possible decompositions of $Sel_{\mathcal{R}}(P)$.
2. Estimate the accuracy of each decomposition.
3. Return the most accurate one.

However, this approach is prohibitively expensive given the large space of possible decompositions, which means that it is necessary to prune this space. In Section 5.2.1, we discuss the approach proposed in [14] for enumerating and pruning this search space

by leveraging some properties of conditional selectivity values. Section 5.2.2 outlines two methods to estimate the accuracy of a given decomposition. Finally, Section 5.2.3 presents the dynamic programming algorithm used in our prototype that returns the most accurate decomposition for a given selectivity estimate.

5.2.1 Enumerating and Pruning Decompositions

This section introduces the notion of "separability" of a decomposition. The separability property can be seen as an indicator of independence that allows simplifying a selectivity value whenever certain properties hold, and can further reduce the search space without missing any relevant decompositions.

Definition 5.4. $Sel_{\mathcal{R}}(P|Q)$ is said to be *separable* (with Q possibly empty) if there exist non-empty sets X_1 and X_2 such that $P \cup Q = X_1 \cup X_2$ and $tables(X_1) \cap tables(X_2) = \phi$. In this case, it is said that X_1 and X_2 separate $Sel_{\mathcal{R}}(P|Q)$.

This definition means that a selectivity estimate $Sel_{\mathcal{R}}(P|Q)$ is separable if all the predicates in P and Q can be split into two sets of predicates X_1 and X_2, where the tables references by these two sets do not overlap. The most intuitive example is when $\sigma_{P \wedge Q}(\mathcal{R}^{\times})$ combines some tables in $\mathcal{R}$ by using Cartesian products, with no join predicates. Note that even if the original query does not use any Cartesian product, after applying atomic decompositions some factors might become separable.

Example 5.4. Consider the non-separable expression $Sel_{\{R,S\}}(R.a < 10, S.b > 5, R.x = S.y)$. After applying an atomic decomposition, we get $Sel_{\{R,S\}}(R.x = S.y|R.a < 10, S.b > 5) \cdot Sel_{\{R,S\}}(R.a < 10, S.b > 5)$, whose second factor is separable.

Property 5.2. (SEPARABLE DECOMPOSITION) [14]. Suppose that $\{P_1, P_2\}$ and $\{Q_1, Q_2\}$ are partitions of P and Q , and $X_1 = P_1 \cup Q_1$ and $X_2 = P_2 \cup Q_2$ separate $Sel_{\mathcal{R}}(P|Q)$. Let $\mathcal{R}_1 = tables(X_1)$ and $\mathcal{R}_2 = tables(X_2)$. Then:

$$Sel_{\mathcal{R}}(P|Q) = Sel_{\mathcal{R}_1}(P_1|Q_1) \cdot Sel_{\mathcal{R}_2}(P_2|Q_2) \tag{5.3}$$

Example 5.5. Since $\{T.b = 5\}$ and $\{R.x = S.y, S.a < 10\}$ separate $s = Sel_{\{R,S,T\}}(T.b = 5, S.a < 10|R.x = S.y)$, we can rewrite s as $s = Sel_{\{R,S\}}(S.a < 10|R.x = S.y) \cdot Sel_{\{T\}}(T.b > 5)$. Here, both resulting factors are no longer separable.

The separable decomposition property is a formal way of describing independence between predicates. The work in [14] makes the assumption that if a selectivity estimate is separable, then computing it as a product of its factors does not affect its accuracy. This has a direct implication on the search space. Using this assumption, it is safe to prune any decompositions that include factors that are separable, without missing the most accurate decomposition.

Generally, there is always a unique decomposition of $Sel_{\mathcal{R}}(P)$ into non-separable factors of the form $Sel_{\mathcal{R}_i}(P_i)$. That is, if we start with $Sel_{\mathcal{R}}(P)$ and repeatedly apply separable decompositions until no single resulting factor is separable, we always obtain the same non-separable decomposition of $Sel_{\mathcal{R}}(P)$. This decomposition is referred to in [14] as the *standard decomposition* of $Sel_{\mathcal{R}}(P)$.

5.2.2 Accuracy Estimation

This section discusses the notion of error, which measures the (estimated) accuracy of a decomposition for given statistics.

Definition 5.5. Let $s = Sel_{\mathcal{R}}(p_1, ..., p_n)$ be a selectivity value, and $S = S_1 \cdot ... \cdot S_k$ be a decomposition of s, where $S_i = Sel_{\mathcal{R}_i}(P_i|Q_i)$. If we use the available statistics to estimate S_i, then $error(S_i)$ measures the estimated accuracy in computing S_i. The value $error(S_i)$ is a positive real number, where smaller values represent better accuracy. The (estimated) overall error for $S = S_1 \cdot ... \cdot S_k$ is given by an aggregate function $E(e_1, ..., e_n)$, where $e_i = error(S_i)$.

In order to follow the principle of optimality, the aggregate function E has to be monotonic. That is, if $x_i \leq x'_i$ for all i, it follows that $E(x_1, ..., x_n) \leq E(x'_1, ..., x'_n)$. Monotonicity is a reasonable property for functions measuring overall accuracy [16, 24, 25]: if each

error e'_i is at least as high as e_i, then the overall error $E(e'_1, ..., e'_n)$ should be at least as high as $E(e_1, ..., e_n)$. This allows a dynamic programming approach to find the most accurate decomposition of $Sel_{\mathcal{R}}(P)$ by trying all atomic decompositions $Sel_{\mathcal{R}}(P) = Sel_{\mathcal{R}}(P_1|P_2) \cdot Sel_{\mathcal{R}}(P_2)$, recursively obtaining the most accurate decomposition of $Sel_{\mathcal{R}}(P_2)$, and combining the partial results.

Note that there are no guarantees on the accuracy of the error estimates given by the error function $error(S_i)$. It is impossible to have an accurate error value without having the actual selectivity values, as explained in Observation 4.2 (Chapter 4). If there is a way to obtain accurate error values, then these error values can be combined with the estimated selectivity values to obtain more accurate results in the first place. But since this is not the case, then we have to be aware that the error function is merely a coarse indicator of accuracy, and cannot be used for any other purpose. However, the "coarseness" of this function can vary depending on how the error function is defined, as shown in the following subsections.

This coarseness is usually also related to the efficiency and ease of evaluating the error function. Functions that give more accurate results tend to be more computationally expensive. When choosing an error function, it is important to choose an efficient one since it is expected to be invoked repeatedly for all enumerated decompositions. Very accurate but inefficient error functions are not useful, since the overall optimization time would increase and therefore exploiting statviews would become less attractive.

In the next two subsections, we describe two different error functions that vary in terms of efficiency and accuracy: The first simple error function *nInd* is introduced in [13], and the second error function *Diff* is adopted in [14].

1. Number of Independence Assumptions (*nInd*)

The first error function *nInd* (adapted from [13]) is simple and intuitive. This function focuses on the error that occurs when the optimizer assumes independence while estimating

the selectivity of a group of conjunctive predicates. *nInd* captures this by counting the number of independence assumptions made by the optimizer.

Suppose that the selectivity value to be estimated is $S = Sel_{\mathcal{R}_1}(P_1|Q_1) \cdot ... \cdot Sel_{\mathcal{R}_n}(P_n|Q_n)$. Assume that each factor $Sel_{\mathcal{R}_i}(P_i|Q_i)$ is estimated using a statview $SV_{R_i}(A'_i|Q'_i)$, where $Q'_i \subseteq Q_i$ and $A'_i \supseteq attr(P_i)$. In this case, the error in estimating S can be defined as the total number of independence assumptions in the estimation. In other words:

$$nInd(S_1, ..., S_n) = \sum_{i=1}^{n} |P_i| \cdot |Q_i - Q'_i| \tag{5.4}$$

where each term above represents the fact that P_i and $Q_i - Q'_i$ are assumed to be independent with respect to Q_i, and therefore the number of independence assumptions is given by $|P_i| \cdot |Q_i - Q'_i|$. For instance, given a statview $SV_{\mathcal{R}}(p|q_1)$, the error $nInd(Sel_{\mathcal{R}}(p|q_1, q_2))$ is equal to 1 (i.e., one independence assumption). It is easy to see that *nInd* is monotonic (according to Definition 5.5, $e_i = |P_i| \cdot |Q_i - Q'_i|$ and E is the sum operator).

The main advantage of *nInd* is that it is very simple and very easy (and efficient) to compute. However, it suffers from the fact that it is a syntactic indication of independence and does not rely on actual correlation measures. Thus it results in very rough error estimates.

2. Difference of Distributions (*Diff*)

Due to the coarseness of the *nInd* metric, many alternatives often result in the same *nInd* value, and ties need to be broken arbitrarily. This behavior is problematic when there are two or more available statviews that can be used to estimate a selectivity value, and while they result in the same "syntactic" *nInd* score, the actual benefit (accuracy) of using each one of them is drastically different, as illustrated in the following example:

Example 5.6. Consider the following query:

```
SELECT * FROM R, S, T
WHERE R.s = S.s
AND S.t = T.t
AND S.a < 10
```

where both joins are defined between primary and foreign keys. Also consider the following decomposition factor that needs to be estimated: $S_1 = Sel_{\{R,S,T\}}(S.a < 10 | R.s = S.s, S.t = T.t)$. Suppose that the only candidates to approximate S_1 are:

- $v_1 = SV_{\{R,S\}}(S.a | R.s = S.s)$, and
- $v_2 = SV_{\{S,T\}}(S.a | S.t = T.t)$

If ***nInd*** is used, both statviews would result in the same error value of 1, so in general each alternative would be arbitrarily chosen. However, v_1 is a much better choice than v_2. In fact, since $S \bowtie_{S.t=T.t} T$ is a foreign-key join, the distribution of $S.a$ over the result of $S \bowtie_{S.t=T.t} T$ is exactly the same as the distribution of $S.a$ over base table S. Therefore, $S \bowtie_{S.t=T.t} T$ is actually independent of $S.a < 10$, and v_2 provides no benefit over the base table statistics over table S.

The ***Diff*** error function [14] overcomes this problem by taking into account the distribution of the required columns in the statviews and how different they are from those in the base tables. In order to be able to compute the ***Diff*** value, additional metadata needs to be maintained for each statview. For a statview $v = SV_{\mathcal{R}}(a_1, ..., a_n | Q)$, a single value ***diff***$_i \in [0, 1]$ has to be maintained for every column a_i in v, that measures the discrepancy between the distribution of a_i in the statview and that of a_i in the base tables it originates from. In particular, ***diff***$_i = 0$ when the two distributions are the same, and ***diff***$_i$ grows up to 1 when such distributions are very different. Generally, there are multiple possible distributions for which ***diff***$_i = 1$, but only one for which ***diff***$_i = 0$.

Consider $v = SV_{\mathcal{R}}(a_1, ..., a_n | Q)$, and suppose that column a_i originates from base table $R_i \in \mathcal{R}$. The value ***diff***$_i$ is defined as follows[1]:

[1]A similar metric, μ_{count}, is proposed in [27] to compare two histogram distributions.

$$diff_i = \frac{1}{2} \cdot \sum_{x \in dom(a)} \left(\frac{|f(R_i, x)|}{|R_i|} - \frac{|f(v, x)|}{|v|} \right) \tag{5.5}$$

where $f(R_i, x)$ and $f(v, x)$ are the frequencies of value x in base table R_i and statview v respectively. The value $diff_i$ measures the deviation of frequencies between the base table and the statview distributions. The $diff_i$ values need to be updated every time the statistics on v are updated, so there is no overhead at runtime.

Using *diff* values, the *Diff* error function provides a less syntactic notion of independence. Suppose that the selectivity value to be estimated is $S = Sel_{\mathcal{R}_1}(P_1|Q_1) \cdot ... \cdot Sel_{\mathcal{R}_n}(P_n|Q_n)$, and assume that each factor $Sel_{\mathcal{R}_i}(P_i|Q_i)$ is estimated using a histogram over column a_i (associated with value $diff_i$) in statview $SV_{R_i}(A'_i|Q'_i)$, where $Q'_i \subseteq Q_i$ and $a_i \subseteq A'_i$. The overall error value of estimated S can be computed as:

$$Diff(S_1, ..., S_n) = \sum_{i=1}^{n} |P_i| \cdot (1 - diff_i) \tag{5.6}$$

The term $(1 - diff_i)$ above represents the degree of independence when estimating S_i using the statistics over column a_i in a statview. This term replaces the "syntactic" value $|Q_i - Q'_i|$ of *nInd*. In Example 5.6, v_2 would result in a *diff* value of 0 since it effectively contributes the same as a base table statistics over column a, which results in the error function being equal to 1 (the maximum possible value). In contrast, the more different the distributions of $S.a$ on S and on v_1, the more likely that v_1 includes dependencies between $S.a$ and $\{R.s = S.s, S.t = T.t\}$, which results in a lower overall error value.

Diff is just a heuristic ranking function and has some natural limitations. For example, it uses a single number ($diff_i$) to summarize the amount of divergence between two distributions. In addition, it only supports predicates on columns from base tables, and not predicates with arbitrary expressions (since these arbitrary expressions do not exist in base tables, and hence do not have distributions that can be compared to those in the statviews). Also, as pointed out earlier, using *Diff* requires extra overhead during statistics collection, since the *diff* values need to be updated.

5.2.3 Selecting the Best Estimate

In this section, we outline a dynamic programming algorithm that obtains the most accurate estimation of $Sel_{\mathcal{R}}(P)$ for a given error function. The algorithm relies on the error function being monotonic, and avoids considering decompositions with separable factors (see Sections 5.2.1 and 5.2.2).

The function *getSelectivity* is shown in Algorithm 5.1. It uses an in-memory look-up table to store the selectivity and error values for previously encountered predicate groups, to avoid repeating computations unnecessarily. Taking as input a set of tables $\mathcal{R}$ and a set of conjunctive predicates P on $\mathcal{R}^{\times}$, the first step (lines 2-3), is to test whether the desired selectivity value was previously calculated, and if so the algorithm returns it using a lookup in the memo table. Otherwise, lines 5-8 handle the case in which $Sel_{\mathcal{R}}(P)$ is separable. Lines 5-6 obtain the standard decomposition of $Sel_{\mathcal{R}}(P)$ and recursively call *getSelectivity* for each factor $Sel_{\mathcal{R}_i}(P_i)$. Then, lines 7-8 combine the partial results. Otherwise (if $Sel_{\mathcal{R}}(P)$ is non-separable), lines 10-20 evaluate all atomic decompositions of $Sel_{\mathcal{R}}(P) = Sel_{\mathcal{R}}(P'|Q) \cdot Sel_{\mathcal{R}}(Q)$. For that purpose, line 12 recursively obtains the most accurate estimation (and the corresponding error) for $Sel_{\mathcal{R}}(Q)$ and line 13 locally obtains the best statview v to estimate $Sel_{\mathcal{R}}(P'|Q)$ among the set of available statviews. If no statviews are available for estimating $Sel_{\mathcal{R}}(P'|Q)$, the algorithm sets $error_{p|Q} = \infty$ and continues with the next atomic decomposition. Lines 14-17 keep track of the most accurate decomposition for $Sel_{\mathcal{R}}(P)$, and after exploring all atomic decompositions, lines 19-20 obtain the most accurate estimation for $Sel_{\mathcal{R}}(P)$. In all cases, before returning $Sel_{\mathcal{R}}(P)$ and its associated error in line 23, *getSelectivity* stores these values in the memo table. Note that a side effect of invoking *getSelectivity(R, P)* is getting the most accurate selectivity estimation for every sub-query $\sigma_{P'}(\mathcal{R}^{\times})$ with $P' \subseteq P$. In Section 5.5 we exploit these “free” selectivity estimates when integrating *getSelectivity* with existing optimizers.

Note that, in line 13, the algorithm attempts to find the best statview v to estimate $Sel_{\mathcal{R}}(P'|Q)$. This is accomplished in two steps:

1. Finding all statviews that are relevant to the current selectivity estimate $Sel_{\mathcal{R}}(P'|Q)$.

Algorithm 5.1 Selecting the most accurate selectivity estimate

```
 1: function GETSELECTIVITY(R, P)
 2:     if Sel_R(P) was already calculated then
 3:         (Sel_R(P), error_P) ← memo_lookup(P)
 4:     else if Sel_R(P) is separable then
 5:         Sel_R1(P_1) · ... · Sel_Rn(P_n) ← standardDecomposition(Sel_R(P))
 6:         (S_Pi, error_Pi) ← getSelectivity(R_i, P_i), i = 1..n
 7:         S_P ← S_P1 · ... · S_Pn
 8:         error_P ← E_merge(error_P1, ..., error_Pn)
 9:     else  // non-separable
10:         error_P ← ∞; v_best ← null
11:         for all P' ⊆ P, Q = P − P' do // atomic decomposition Sel_R(P'|Q) · Sel_R(Q)
12:             (S_Q, error_Q) ← getSelectivity(R, Q)
13:             (v, error_P'|Q) ← best statview and error for Sel_R(P'|Q)
14:             if E_merge(error_P'|Q, error_Q) ≤ error_P then
15:                 error_P ← E_merge(error_P'|Q, error_Q)
16:                 v_best ← v
17:             end if
18:         end for
19:         S_P'|Q ← estimation of Sel_R(P'|Q) using v_best
20:         S_P ← S_P'|Q · S_Q
21:     end if
22:     memo_insert(P, S_P, error_P)
23:     return (S_P, error_P)
24: end function
```

This is the view matching part of exploiting statviews, and the matching conditions are described in more detail in Section 5.4.

2. Computing the error function based on each of these statviews, and returning the one with the least estimated error.

Algorithm $getSelectivity(\mathcal{R}, P)$ returns the most accurate estimation of $Sel_{\mathcal{R}}(P)$ for a given definition of error among all non-separable decompositions.

5.3 Estimation using Maximum Entropy

The conditional selectivity approach estimates a selectivity value by considering the different possible ways of estimating such value (different compositions), and estimates the resulting error in each case, then picks the most accurate estimate (the one with the least estimated error value. Although this approach guarantees consistency in estimation, it still has some shortcomings: First, the estimated error might not be accurate itself, and might lead the optimizer to choose a decomposition that is not the most accurate. Second, by using only one decomposition, the optimizer only incorporates knowledge from a subset of the available statviews. Possibly ignoring some knowledge can bias the optimizer towards choosing one particular plan over another.

In general, an optimizer will often be drawn towards those plans about which it knows the least, because using the independence assumption makes these plans seem cheaper due to underestimation. This problem is often referred to as “fleeing from knowledge to ignorance” [42].

In this section, we present a different approach, inspired by the work in [42]. This approach exploits and combines all of the available statistics (base table statistics as well as statviews) in a principled, consistent, and unbiased manner to estimate the selectivity of a set of predicates. The technique is based on the principle of maximum entropy (ME)

(cf. Section 2.4), which provides the "simplest" possible selectivity estimate that is consistent with all of the available information. In the absence of detailed knowledge, the ME approach reduces to standard uniformity and independence assumptions. This approach avoids the problems of inconsistent plan comparisons and the flight from knowledge to ignorance.

For this approach, a statview $v = SV_{\mathcal{R}}(A|P)$ provides the following statistics:

- The number of rows $|v|$ can be used to estimate the selectivity $Sel_{\mathcal{R}}(P)$ (given that $|\mathcal{R}^{\times}|$ can be easily computed).
- Statistics on every attribute $a \in A$ can be used to estimate the selectivity $Sel_{\mathcal{R}}(q, P)$ for any predicate q where $attr(q) = a$.

Example 5.7. Consider the statview $v = SV_T(a, b|c < 10, d = 2)$, with histograms over the columns a and b. This statview can be used to compute the following selectivities:

- $Sel_T(c < 10, d = 2)$ can be estimated as $|v|/|T|$.
- $Sel_T(a > 3, c < 10, d = 2)$ can be estimated by obtaining the number of tuples that satisfy $a > 3$ from the histogram on a, and dividing it by $|T|$. The same applies if the first predicate is replaced by any other predicate on a or b.

The following subsections discuss the details of this approach. Section 5.3.1 redefines the selectivity estimation problem in a way that can be approached using the maximum entropy principle. Section 5.3.2 defines the *constrained optimization problem* and how it can be used in this approach. Finally, Section 5.3.3 presents our high-level algorithm that puts together all the concepts used in this approach.

5.3.1 Formal Definition

In this section, we formalize the problem of selectivity estimation for conjunctive predicates, given partial statistics, and define some useful terminology. Let $P = \{p_1, ..., p_n\}$ be

a set of predicates, and let $N = \{1, ..., n\}$. For any $X \subseteq N$, let p_X denote the conjunction $\wedge_{i \in X} p_i$. For example, if $X = \{1, 3, 6\}$ then p_X denotes $p_1 \wedge p_3 \wedge p_6$. Any given assignment of X corresponds to a subset of the predicates in P. Let s_X be the combined selectivity of p_X for a given X. For $|X| = 1$ (single predicates), the histograms and column statistics available on base tables are sufficient to estimate s_X. For $|X| > 1$, the multi-variate statistics (MVS) may be stored in the system catalog either as multidimensional histograms, index statistics, or some other form of column-group statistics or statviews. In practice, s_X is not known for all possible predicate combinations due to the exponential number of combinations of columns that can be used to define the MVS.

The powerset of N, denoted by 2^N, is the set of all possible assignments of X, i.e. all possible predicate combinations. Let $T \subset 2^N$ be the set of predicate combinations for which s_X is known[2]. Then the selectivity estimation problem is to compute s_X for $X \in 2^N$ given T.

A key aspect of this approach is that the query optimizer should avoid any additional assumptions about the unknown selectivities while simultaneously exploiting all existing knowledge in order to avoid unjustified bias towards any particular solution. Applying the maximum entropy principle to selectivity estimation means that, given several selectivities of simple predicates and conjuncts, the optimizer should choose the most uniform/independent selectivity model consistent with all of this knowledge.

5.3.2 The Constrained Optimization Problem

In this section, we describe the technique used in [42] to solve the selectivity estimation problem. For a predicate p, let p^1 indicate the predicate p itself, and p^0 indicate the negation of p, i.e. $p^1 = p$ and $p^0 = \neg p$. An *atom* is a term in disjunctive normal form (DNF) over the space of n predicates, i.e., a term of the form $p_1^{b_1} \wedge ... \wedge p_n^{b_n}$ for $b_i \in \{0, 1\}$. We denote this atom by the vector $b = (b_1, ..., b_n) \in \{0, 1\}^n$. As a further abbreviation, we sometimes omit the parentheses and commas when denoting a specific atom.

[2]Note that the empty set ϕ is part of T, as $s_\phi = 1$ when applying no predicates.

Example 5.8. Given $P = \{p_1, p_2, p_3\}$ with $|P| = 3$, the string 100 denotes the vector $(1, 0, 0)$ and thus the atom $p_1 \wedge \neg p_2 \wedge \neg p_3$.

For a predicate group $p_X, X \in 2^N$, let $C(X)$ denote the set of components of X, i.e, the set of all atoms contributing to p_X. Formally,
$C(X) = \{b \in \{0,1\}^n | \forall i \in X : b_i = 1\}$ and
$C(\phi) = \{0,1\}^n$

Example 5.9. Given $P = \{p_1, p_2, p_3\}$, for $X = \{1\}$, the predicate group p_1 (having a single predicate in this case), can be expressed as:
$p_1 = (p_1 \wedge \neg p_2 \wedge \neg p_3) \vee (p_1 \wedge p_2 \wedge \neg p_3) \vee (p_1 \wedge \neg p_2 \wedge p_3) \vee (p_1 \wedge p_2 \wedge p_3)$.
Thus, we can write: $C(\{1\}) = \{100, 110, 101, 111\}$. Similarly, for $X = \{1, 2\}$, we have $p_{1,2} = (p_1 \wedge p_2 \wedge \neg p_3) \vee (p_1 \wedge p_2 \wedge p_3)$. Thus, we can write: $C(\{1, 2\}) = \{110, 111\}$.

Additionally, for a set $T \subseteq 2^N$ of known predicate combinations, and for an atom b, let $P(b, T)$ denote the set of all $X \in T$ such that p_X has b as an atom in its DNF representation, i.e., $P(b, T) = \{X \in T | \forall i \in X : b_i = 1\} \cup \{\phi\}$.

Example 5.10. Using the same set P from the previous two examples, suppose that $T = \{\{1\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \phi\}$. For the atom $b = 011$, this atom is a component of the predicate groups p_3, $p_{2,3}$, and p_ϕ. Hence, we obtain the set $P(b, T) = \{\{3\}, \{2, 3\}, \phi\}$.

Let x_b denote the selectivity of an atom b, with $x_b \geq 0$. Given s_X for $X \in T$, we want to compute s_X for $X \notin T$ according to the maximum entropy principle. To achieve this objective, we must solve the following *constrained optimization problem* [42]:

Definition 5.6. (CONSTRAINED OPTIMIZATION PROBLEM) Given the $|T|$ constraints:

$$\forall X \in T, \sum_{b \in C(X)} x_b = s_X \tag{5.7}$$

it is required to maximize the entropy, i.e. minimize $\sum_{b \in \{0,1\}^n} x_b \cdot log(x_b)$

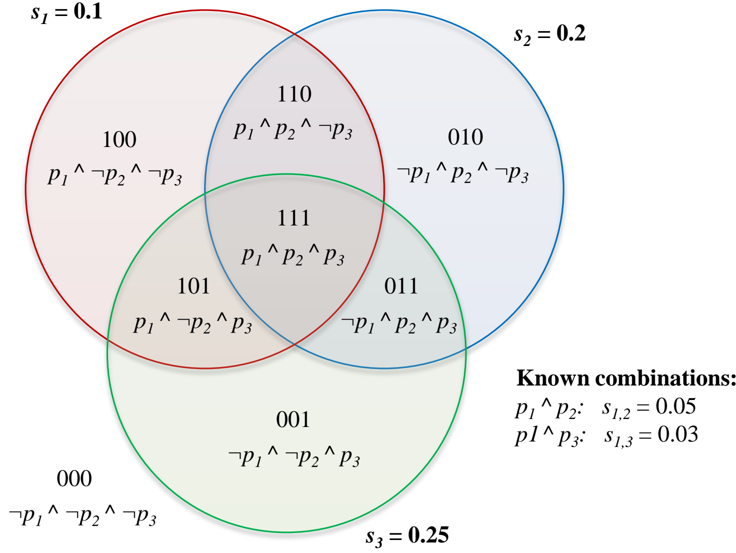


Figure 5.1: Probability space for $|T| = 6$ and $N = \{1, 2, 3\}$

The given constraints are the known selectivities, and the solution is a probability distribution with the maximum value of uncertainty (entropy), subject to the constraints. One of the included constraints is $s_\phi = \sum_{b \in \{0,1\}^n} x_b = 1$, which asserts that the combined selectivity of all atoms is 1. The above problem analytically can be solved analytically only in simple cases with a small number of unknowns. In general, a numerical method is required.

Example 5.11. Figure 5.1 shows the probability space for the predicate space created by $N = \{1, 2, 3\}$ and the knowledge set $T = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \phi\}$ with the selectivities $s_1 = 0.1$, $s_2 = 0.2$, $s_3 = 0.25$, $s_{1,2} = 0.05$, $s_{1,3} = 0.03$, and $s_\phi = 1$.

This example results in the following six constraints:

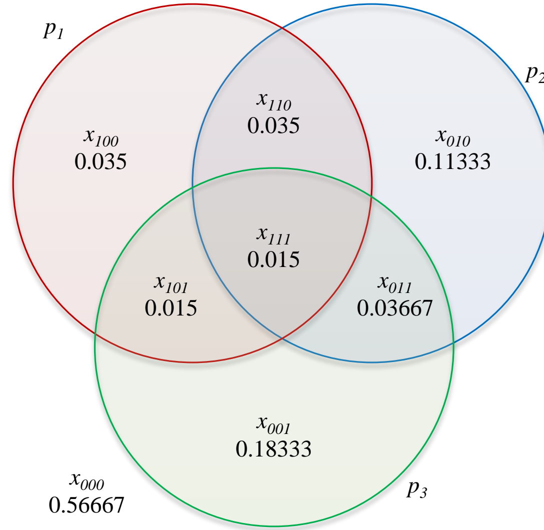


Figure 5.2: Maximum entropy solution

$$
\begin{aligned}
s_1 &= x_{100} + x_{110} + x_{101} + x_{111} &&= 0.1\\
s_2 &= x_{010} + x_{011} + x_{110} + x_{111} &&= 0.2\\
s_3 &= x_{001} + x_{011} + x_{101} + x_{111} &&= 0.25\\
s_{1,2} &= x_{110} + x_{111} &&= 0.05\\
s_{1,3} &= x_{101} + x_{111} &&= 0.03\\
s_{\phi} &= \sum_{b \in \{0,1\}^3} x_b &&= 1
\end{aligned}
$$

The task of selectivity estimation is now to compute a solution for all atoms $x_b, b \in \{0,1\}^3$ that maximizes the entropy function: $-\sum_{b \in \{0,1\}^n} x_b \cdot log(x_b)$ and satisfies the above six constraints. Once this is achieved, it would be possible to compute all $s_i, i \in 2^{\{1,2,3\}}$, from the x_b using Equation 5.7.

Figure 5.2 gives the results obtained when solving this constrained optimization problem. For example, in this solution, we obtain the selectivity estimate $s_{1,2,3} = x_{111} = 0.015$ and $s_{2,3} = x_{111} + x_{011} = 0.05167$.

The problem now is solving the system of simultaneous equations to obtain the atoms x_b, in a way that maximizes the entropy. As mentioned earlier, solving these equations analytically is only possible when there is a small number of unknowns. We use the algorithm proposed in [42] to compute this result efficiently for an arbitrary number n of simple predicates P and an arbitrary set T of constraints.

This method uses a variant of the iterative scaling algorithm [21] to efficiently obtain an approximate maximum entropy solution. This is achieved by using Lagrange multipliers to obtain a system of optimality equations. The algorithm them iteratively refines those multipliers until they converge to their final values. We do not list the details of the algorithm since it is not key to our work, but the full details can be found in [42].

5.3.3 Computing the Selectivity Estimate

Our algorithm to compute the selectivity estimate is given by Algorithm 5.2. The function *getSelectivityME* takes as inputs the set of predicates P and the set of tables $\mathcal{R}$, and returns the selectivity value $Sel_{\mathcal{R}}(P)$.

The function first checks to see if there exists a statview (or base table statistic) that can give the required estimate directly, in which case, it would just use that statview to estimate the required selectivity and return it (lines 2-6). If no such statview is found, the function then proceeds to determine which combinations of the predicates in P have known selectivities. To accomplish this, it iterates over all possible subsets of P. For each subset P', the algorithm attempts to locate a statview (or base table statistic) that can produce the estimate $Sel_{\mathcal{R}}(P')$ (line 9). If such an estimate can be computed, then the set X is formed based on the predicates in P' (line 12). For example, if $P' = \{p_1, p_4\}$, and $Sel_{\mathcal{R}}(P')$ can be directly estimated from a statview, then $X = \{1, 4\}$. The set X and the estimated selectivity value for P' are then added to the set of constraints T (line 13). As a final step, the empty set ϕ and its selectivity of 1 are also added to T, then the iterative scaling algorithm is called (lines 16-17). The iterative scaling algorithm returns an array x. Each element in this array is the selectivity of an atom x_b. Finally, the required selectivity of P is the selectivity of the atom $x_{111...1}$ (where all predicates are present).

Algorithm 5.2 Selectivity estimation using maximum entropy

```
1:  function getSelectivityME(R, P)
    Assume that P = {p_1, p_2, ..., p_n}
2:      v ← statview that can directly give Sel_R(P)
3:      if (v ≠ null) then
4:          S ← estimation of Sel_R(P) using v
5:          return S
6:      end if
7:      T ← {}
8:      for all P' ⊂ P do
9:          v ← statview that can directly give Sel_R(P')
10:         if (v ≠ null) then
11:             s_X ← estimation of Sel_R(P') using v
12:             X ← {i|p_i ∈ P'}
13:             T.add(X, s_X)
14:         end if
15:     end for
16:     T.add(φ, 1)
17:     x ← iterativeScaling(T, |P|)
18:     S = x_111...1
19:     return S
20: end function
```

Note that finding a statview that can provide a particular estimate, e.g. lines 2 and 9, is achieved using view matching, which is discussed in Section 5.4.

5.4 Statview Matching Conditions

As seen in Algorithms 5.1 and 5.2, at some point it is required to find a statview (or statviews) that can be used to provide a particular estimate. This task is known as view

matching, and is similar to the view matching techniques used with materialized views. (cf. Section 2.2.2). However, as previously noted in Section 2.3.4, the view matching conditions that are used for materialized views are not suitable to be used with statviews. As a result, in the section, we discuss the view matching that we use for view matching in our work. We list the matching conditions, as well as any limitations that are imposed by the conditional selectivity and the maximum entropy approaches.

5.4.1 Matching for the Conditional Selectivity Approach

For the conditional selectivity approach, a statview $v = SV_{\mathcal{T}}(A|P')$ is considered a successful match to obtain the selectivity estimate $Sel_{\mathcal{R}}(P|Q)$ if all the following conditions are met:

1. The table sets $\mathcal{R}$ and $\mathcal{T}$ must have the same tables. However, it is also acceptable if there are extra tables in either of them as long as they do not change the distribution of the data (e.g. using a foreign key to primary key join).
2. The predicates in P' must be a subset of (or equal to) those in Q. The independence assumption is made for the remaining predicates, affecting the resulting estimated error value.
3. All attributes involved in the predicates P must belong to one of the histograms defined on v. This is important to be able to use that histogram to estimate the required selectivity. Since the prototype does not support multi-dimensional histograms, $|attr(P)|$ must be equal to 1, and $attr(P) \subset A$.

The required selectivity is estimated using the standard estimation techniques. Since it is a conditional selectivity value, it is computed relative to the number of tuples in the statview, not relative to the original base tables. Hence, the optimizer does not need any information about the original base tables other than what is included with the statview.

Example 5.12. Consider the selectivity estimate $Sel_{\{R,S\}}(R.y = 5|R.s = S.s, R.x < 10)$. The following statviews would both be considered successful matches:

- $v_1 = SV_{\{R\}}(R.y, R.z|R.x < 10)$ is a good match since the missing table S does not affect the distribution of $R.y$ (foreign key join).
- $v_2 = SV_{\{R,S\}}(R.y, S.a|R.s = S.s, R.x < 10)$ is a good match since the predicates match exactly, and $R.y$ is in the statview's output columns.

The following statviews would *not* match the given selectivity estimate:

- $v_3 = SV_{\{R\}}(R.z|R.x < 10)$ cannot be used since $R.y$ is not one of the output columns of v_3.
- $v_4 = SV_{\{R,S\}}(R.y, S.a|R.s = S.s, R.z = 2)$ cannot be used since it has an extra predicate $R.z = 2$.

Using the conditional selectivity approach, there are some limitations on what statviews can be defined, and what query expressions can be matched. These limitations include the following:

- When using the *Diff* error function, the set A can only have single columns and not arbitrary arithmetic expressions, since the *Diff* function cannot be defined over arbitrary expressions. This is not an issue with the *nInd* error function, since it only considers the number of independence assumptions without looking at the predicate structure or the output expressions of the statview.
- As noted earlier, the predicate set P must contain predicates that reference only one attribute, since the prototype does not support multi-dimensional histograms. However, this is an implementation limitation. If the system supports multi-dimensional histograms, then P can have predicates on multiple attributes. A possible alternative is to allow P to have predicates on multiple attributes as long as $attr(P) \subset A$. In

this case, the selectivity of each predicate can be estimated from its corresponding histogram, and then they can be combined assuming independence. This would have an effect on the associated error function.

5.4.2 Matching for the Maximum Entropy Approach

Using the maximum entropy approach, a statview $v = SV_{\mathcal{T}}(A|P')$ is considered a successful match to obtain the selectivity estimate $Sel_{\mathcal{R}}(P)$ if all the following conditions are met:

1. The table sets $\mathcal{R}$ and $\mathcal{T}$ must have the same tables. However, it is also acceptable if there are extra tables in either of them as long as they do not change the distribution of the data (e.g. using a foreign key to primary key join).
2. The predicates in the view definition must be a subset of, or equal to, the predicates in the required selectivity estimate, i.e. $P' \subseteq P$
3. If $P' \subset P$ (not equal), then all attributes involved in the predicates $P - P'$ must belong to one of the histograms defined on v, for the same reason as in the conditional selectivity approach.

If the two predicate groups P and P' are identical, then the selectivity $Sel_{\mathcal{R}}(P)$ can be computed as $|v|/|\mathcal{R}^{\times}|$. The numerator is among the statistics stored with v, while the denominator is the product of the cardinalities of the tables in $\mathcal{R}$, which are easily obtained from the system catalog. However, for the case where P' is not equal to P, assume that $P' - P$ is the predicate *a relop value*, where $a \in A$. The number of tuples satisfying this predicate can be obtained from the histogram on attribute a (one of the histograms associated with statview v), then divided by the same denominator $|\mathcal{R}^{\times}|$ to obtain the selectivity $Sel_{\mathcal{R}}(P)$.

Example 5.13. Consider the selectivity estimate $Sel_{\{R,S\}}(R.s = S.s, R.y = 5, R.x < 10)$. The statviews $v_1 = SV_{\{R,S\}}(R.y, S.a|R.s = S.s, R.x < 10)$ would be a successful match since the statview's predicates are a subset of the predicates in the selectivity estimate, and the remaining predicate $R.y = 5$ involves only one column, which is in the output columns of v_1. The following statviews would *not* match the given selectivity estimate:

- $v_2 = SV_{\{R,S\}}(R.z, S.a|R.s = S.s, R.x < 10)$ cannot be used since $R.y$ is not one of the output columns of v_2.
- $v_3 = SV_{\{R,S\}}(R.y, S.a|R.s = S.s, R.z = 2)$ cannot be used since it has an extra predicate $R.z = 2$.

This approach also suffers from the same limitation of not having the support for multi-dimensional histograms. Which means that, in our implementation, the predicates in $P - P'$ must reference only one attribute in order to be able to estimate their selectivity. This limitation is not inherent to the approach itself, and can be overcome if the system supports multi-dimensional histograms.

5.5 Integration with the *PostgreSQL* Optimizer

The existing *PostgreSQL* [3] optimizer computes cardinality and cost estimates during plan enumeration and costing. The optimizer distinguishes between cardinality estimates of base relation and those of join expressions. However, both types of estimates are computed using the same estimation routines. When costing a plan that involves a set of conjunctive predicates $P = p_1 \wedge ... \wedge p_n$, the optimizer calls the estimation module for every predicate $p_i \in P$, using available base table statistics to estimate the selectivity of p_i, then combines these selectivities assuming independence.

Our new estimation modules that take into account the existing statviews were straightforward to integrate into the *PostgreSQL* optimizer. For our prototype implementation, we

extended the selectivity estimation functionality as follows: Given an SPJ query expression for which the optimizer needs to obtain a selectivity estimate, the optimizer invokes either the *getSelectivity* (algorithm 5.1) or *getSelectivityME* (Algorithm 5.2) functions, depending on which method is being used, and passes the set of tables and predicates involved in the required query expression. These functions try to compute the required estimate by finding the relevant statviews and using their respective techniques on these statviews, if any. Note that if no statview cover the required expression, both techniques fall back to estimation using only the base table statistics.

Our extensions to the estimation module enable the optimizer to use all available statistics in a consistent way, for all plans in the plan space. This improved knowledge results in better query plans and improved query execution times, as shown experimentally in the next section.

5.6 Experimental Evaluation

In this section, we present experimental results of our implementation of the two techniques outlined in this chapter.

5.6.1 Setup

Computing environment: All our experiments were conducted on a SunFire X4100 server (see Table 5.1). We implemented our prototype in C inside the PostgreSQL [3] database system.

Data: We carried out our experiments on 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,

Table 5.1: Computing environment for the experiments

CPU	Two dual-core AMD Opteron 280 CPUs
RAM	8 GB
Disk	Two 72 GB disks, in RAID-0 configuration
Operating System	OpenSuSE Linux 10.1

whereas others are skewed. A number of correlations between attributes, such as `Make` and `Model`, are inherent in the attribute definitions.

Queries: We used a workload that contains 80 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
```

Statviews: We used our *StatAdvisor* framework (Chapter 4) to recommend statviews for the given workload, without imposing any restrictions on the number or the size of the statviews that we can maintain. The *StatAdvisor* recommended 99 statviews.

Selectivity estimation: We implemented both the conditional selectivity *(CS)* (Section 5.2) and the maximum entropy *(ME)* (Section 5.3) approaches for selectivity estimation. For the conditional selectivity approach, we implemented both the *nInd* and the *Diff* error estimation techniques (Section 5.2.2).

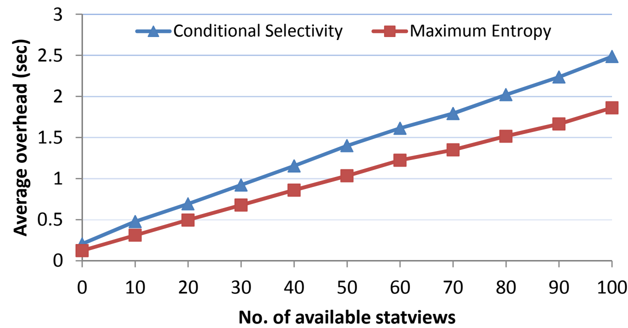


Figure 5.3: Average matching overhead

5.6.2 Statview Matching Overhead

Attempting to use statviews in selectivity estimation creates some additional overhead as a result of the complicated computations as well as view matching. The amount of view matching overhead increases with the number of available statviews in the system, since the optimizer attempts to match all these statviews against every query expression for which it requires a selectivity estimate.

Figure 5.3 depicts the average overhead (over all queries in the workload) for both the conditional selectivity and the maximum entropy approaches, for different number of available statviews. As expected, the overhead increases linearly with the number of available statviews. The overhead in the case of the conditional selectivity approach is greater than that in the case of the maximum entropy approach because the conditional selectivity approach attempts to decompose each query expression into factors and recursively estimate the selectivity of these factors, which means attempting to match these factors against all the available statviews as well.

5.6.3 Workload Performance

The test workload is executed in the following settings:

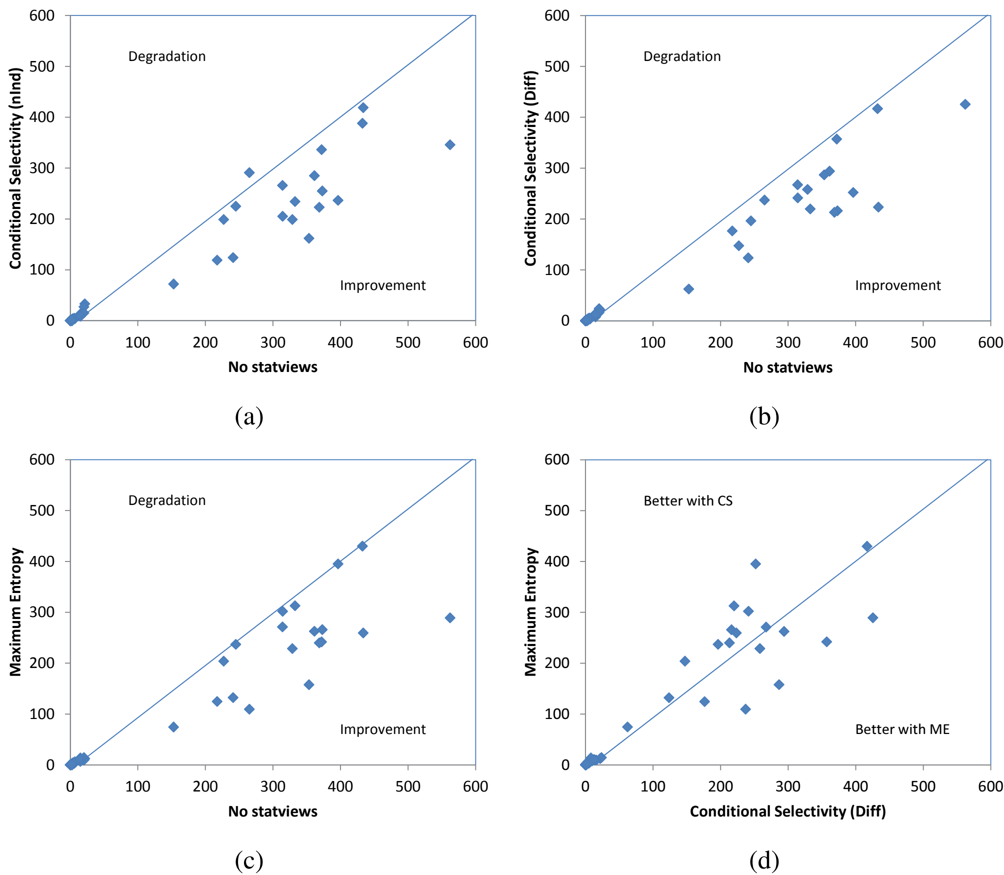


Figure 5.4: Workload Performance

No Statviews The optimizer estimates selectivities using only statistics on base tables and their attributes, including table cardinalities, number of distinct values in each column, etc.

Conditional Selectivity *(nInd)* The optimizer has access to all statviews, and used the *CS* approach with the *nInd* error function to estimate selectivities.

Conditional Selectivity *(Diff)* The optimizer uses the *CS* approach with the *Diff* error function, with all statviews available.

Maximum Entropy The optimizer uses the *ME* approach for selectivity estimation, again

with all statviews available.

Figure 5.4(a) depicts a comparison between the workload performance in the first two settings. Each point represents a single query, with the x-axis representing the execution time of the query when no statviews are used, and the y-axis representing the execution time in the *CS-nInd* setting. It can be seen that most queries benefit when statviews are used in selectivity estimation, since they provide more accurate estimates, helping the optimizer to choose a more efficient execution plan. Similarly, Figures 5.4(b) and 5.4(c) compare the "no statviews" setting to the *CS-Diff* and the *ME* settings, respectively, showing similar results.

Figure 5.4(d) compares the query performance in the *CS* (with the *Diff* error function) and the *ME* settings. The figure shows that the two approaches are comparable, since some queries benefit more from the *CS* approach while others benefit from the *ME* approach.

5.6.4 Estimation Accuracy

In this section we demonstrate the improvement in selectivity estimation accuracy resulting from using statviews. For each query expression encountered during optimization and present in the final execution plan, we record the estimated cardinality and the actual cardinality monitored during query execution. We compute the absolute estimation error as the difference between the estimated and actual cardinality (number of rows).

Figure 5.5(a) shows the improvement in selectivity estimation when using the conditional selectivity approach. Each point corresponds to a query expression, with the x-axis being the absolute estimation error when no statviews are used, and the y-axis being the absolute estimation error when the *CS* approach is used. While some expressions have the same estimation error in both cases (the points lying on the 45-degree line), most expressions have their error reduced significantly by the exploitation of statviews. The same is shown for the maximum entropy approach in Figure 5.5(b).

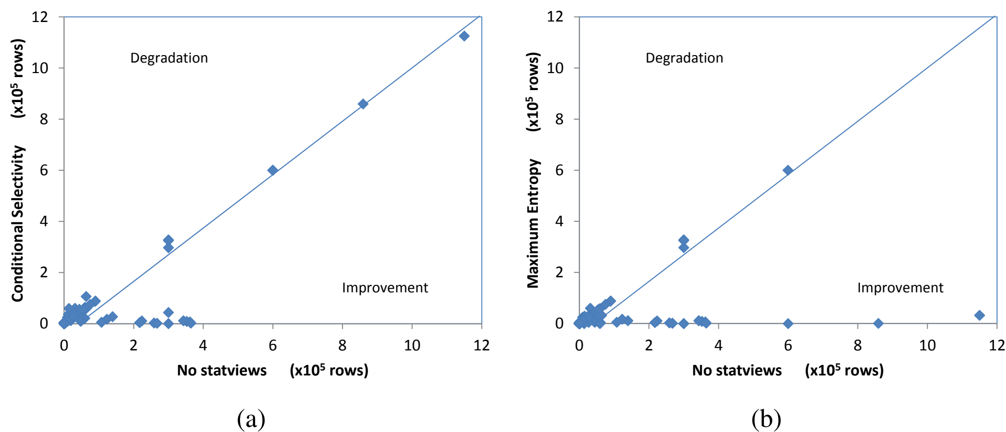


Figure 5.5: Absolute estimation error

5.7 Conclusions

This chapter outlines our implementation of two approaches for exploiting statviews in query optimizations. The conditional selectivity [14] approach analyzes all possible ways of computing a selectivity estimate, and uses the one that has the minimum estimated error, while the maximum entropy approach [42] makes use of all the available information while computing the estimate, without making further assumptions about the data. We implemented the two methods inside the *PostgreSQL* [3] query optimizer and demonstrated their effectiveness with an extensive experimental study.

Lessons Learned. Based on the study and the experiments presented in this chapter, we make the following high level observations:

- Both implemented approaches are comparable in performance, although the conditional selectivity is slightly more expensive dues to its recursive nature. Their effect on workload performance is also similar.
- Statview matching can be computationally expensive, and its cost increases with the number of statviews available in the system. Therefore, it is important not to create too many statviews. Tools such as our *StatAdvisor* can help recommend the most beneficial statviews for a given workload.

- Both approaches are limited regarding the query expressions whose selectivity they can estimate. Both approach work only for SPJ query expressions, and are not straightforward to extend to aggregations or sub-queries for example. In addition, predicates with arbitrary arithmetic expressions can pose a challenge to both approaches. These types of query expressions need further investigation.

Chapter 6

Conclusions and Future Work

In this chapter we conclude this dissertation, and present future research directions.

6.1 Conclusions

This dissertation presents our work on query optimization in dynamic environments. Our study tackles the topics of recommending and collecting database statistics, either on the fly during query processing, or as an offline process in the form of statviews. In addition, we also address the research topic of exploiting statviews for selectivity estimation.

We introduced an efficient approach to proactively determine, collect, and materialize Just-in-Time Statistics (JITS) during query processing. In contrast to similar research efforts, our approach employs a lightweight sensitivity analysis based on the query structure, the existing statistics and the data activity to identify the crucial statistics. The approach also materializes and incrementally updates the collected partial statistics for future reuse. Our technique integrates these partial statistics in a reusable form by maintaining maximum-entropy-based structures.

We proposed *StatAdvisor*, a framework for automatic recommendation of 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. Our work also considers the possible dependency between multiple statviews in terms of their effect on the chosen execution plan.

Finally, we also implemented a prototype for exploiting statviews in query optimizations, using both the conditional selectivity and maximum entropy approaches, applying those methods in the context of statviews.

6.2 Future Work

Possible future extension for our work include investigating new techniques to improve our just-in-time statistics functionality, extending statview exploitation to support arbitrary query expressions rather than only SPJ query expressions, and extending our statview recommendation framework to generate statviews that can benefit multiple queries, thus reducing the number of statviews in the system. In the following sub-sections, we give a high level overview on each of these extensions.

6.2.1 Enhancing the Just-in-Time Statistics Functionality

When collecting just-in-time statistics, we perform a lightweight sensitivity analysis to determine which statistics need to be collected. The used analysis is based on a history of statistics usage in cardinality estimation. It would be interesting to consider a more sophisticated sensitivity analysis that takes into account previously collected QSS and whether or not they caused the optimizer to choose a different plan. This would naturally result in increased overhead for the JITS module, so it would be necessary to study the cases where this would be beneficial.

It would also be interesting to investigate methods to further reduce the time spent on statistics collection during query processing. A possible way to achieve this is to infer missing statistics based on the existing ones (both in the catalog and in the QSS archive). by integrating catalog statistics with sampled data, and/or inferring some of the absent statistics. Inferred statistics can be associated with a confidence score, and this score can be used by the optimizer to decide whether it can rely on these inferred statistics or it is necessary to spend more time to collect more statistics.

6.2.2 Recommending "Generalized" Statviews

In our *StatAdvisor* framework, we determine the statviews that would benefit a particular query and recommend them together as a statview group. There are cases where similar, but not identical, statviews are recommended for separate queries. For example, one query may benefit from having a statview `v1 = Select salary from employee where state='NY' and salary < 100k`, while another query may benefit from having the statview `v2 = Select salary from employee where state='NY' and salary < 150k`. In this particular case, maintaining only $v2$ and having a histogram on the *salary* column would suffice, since $v2$ "covers" $v1$.

Consider another case with the same statview $v1$ above, but with `v2 = Select salary from employee where state='NY' and salary > 150k`. In this case, both statview are not overlapping. It would be useful to generate a "merged" view `v = Select salary from employee where state='NY'`. Maintaining this single view with a histogram on the *Salary* column would provide accurate statistics for both queries. Similar work for finding merged materialized views has been done in [8].

Maintaining one statview instead of multiple statviews saves on the space needed to store the statistics as well as the time required for statview matching during query optimization.

6.2.3 Statview Matching for Complex Query Expressions

In Chapter 5, we explored two techniques to exploit statviews in selectivity estimation. However, both techniques can only be applied for estimating the selectivity of SPJ query expressions. It would be interesting to investigate possible ways to extend these methods, or find new methods that can be applied to other types of query expressions and/or operators. Examples for such expressions include aggregations, grouping, outer joins, and nested queries.

Being able to support arbitrary and complex query expressions greatly would increase the usefulness and applicability of statviews, which in turn would help reduce estimation errors in a variety of queries that are used in analytical applications and decision support systems.

Bibliography

[1] DB2 for Linux, UNIX and Windows, *http://www.ibm.com/software/data/db2/9*. 47, 76

[2] Microsoft SQL Server, *http://www.microsoft.com/sql*. 18

[3] PostgreSQL, *http://www.postgresql.org/*. 87, 111, 112, 117

[4] TPC-DS Benchmark, *http://www.tpc.org/tpcds*. 76

[5] Ashraf Aboulnaga, Peter J. Haas, Mokhtar Kandil, Sam Lightstone, Guy M. Lohman, Volker Markl, Ivan Popivanov, and Vijayshankar Raman. Automated Statistics Collection in DB2 UDB. In *VLDB*, pages 1146–1157, 2004. 4, 14, 47

[6] Mohammed Abouzour, Ivan T. Bowman, Peter Bumbulis, David DeHaan, Anil K. Goel, Anisoara Nica, G. N. Paulley, and John Smirnios. Database Self-Management: Taming the Monster. *IEEE Data Eng. Bull.*, 34(4):3–11, 2011. 14

[7] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join Synopses for Approximate Query Answering. In *SIGMOD*, pages 275–286, 1999. 25, 68

[8] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, pages 496–505, 2000. 18, 62, 121

[9] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, pages 261–272, 2000. 15

[10] Brian Babcock and Surajit Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *SIGMOD*, pages 119–130, 2005. 25, 61

[11] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive Re-optimization. In *SIGMOD*, pages 107–118, 2005. 4, 15

[12] Ivan T. Bowman, Peter Bumbulis, Dan Farrar, Anil K. Goel, Brendan Lucier, Anisoara Nica, G. N. Paulley, John Smirnios, and Matthew Young-Lai. SQL Anywhere: A Holistic Approach to Database Self-management. In *ICDE Workshops*, pages 414–423, 2007. 14

[13] Nicolas Bruno and Surajit Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *SIGMOD*, pages 263–274, 2002. 5, 7, 21, 22, 23, 58, 61, 84, 94

[14] Nicolas Bruno and Surajit Chaudhuri. Conditional Selectivity for Statistics on Query Expressions. In *SIGMOD*, pages 311–322, 2004. 5, 8, 21, 23, 87, 88, 89, 91, 92, 93, 94, 96, 117

[15] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. *SIGMOD Rec.*, 30(2):211–222, 2001. 14

[16] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating Top-k Queries over Web-Accessible Databases. In *ICDE*, pages 369–380, 2002. 93

[17] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On Random Sampling over Joins. In *SIGMOD*, pages 263–274, 1999. 25

[18] Surajit Chaudhuri and Vivek Narasayya. Automating Statistics Management for Query Optimizers. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):7–20, 2001. 15, 22, 34, 57, 58, 70

[19] Surajit Chaudhuri and Moshe Y. Vardi. Optimization of Real Conjunctive Queries. In *PODS*, pages 59–70, 1993.

[20] Francis C. Chu, Joseph Y. Halpern, and Johannes Gehrke. Least Expected Cost Query Optimization: What Can We Expect? In *PODS*, pages 293–302, 2002.

[21] John N. Darroch and Douglas Ratcliff. Generalized Iterative Scaling for Log-Linear Models. In *The Annals of Mathematical Statistics*, volume 43, pages 1470–1480, 1972. 106

[22] Amr El-Helw, Ihab F. Ilyas, Wing Lau, Volker Markl, and Calisto Zuzarte. Collecting and Maintaining Just-in-Time Statistics. In *ICDE*, pages 516–525, 2007. 29, 31

[23] Amr El-Helw, Ihab F. Ilyas, and Calisto Zuzarte. StatAdvisor: Recommending Statistical Views. *PVLDB*, 2(2):1306–1317, 2009. 55

[24] Ronald Fagin. Fuzzy Queries in Multimedia Database Systems. In *PODS*, pages 1–10, 1998. 93

[25] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, pages 102–113, 2001. 93

[26] César A. Galindo-Legaria, Milind Joshi, Florian Waas, and Ming-Chuan Wu. Statistics on Views. In *VLDB*, pages 952–962, 2003. 2, 3, 5

[27] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast Incremental Maintenance of Approximate Histograms. In *VLDB*, pages 466–475, 1997. 96

[28] Jonathan Goldstein and Per-Åke Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, 2001. 17, 20, 21

[29] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[30] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218, 1993.

[31] Silviu Guiasu and Abe Shenitzer. The Principle of Maximum Entropy. *The Mathematical Intelligencer*, 7(1):42–48, 1985. 8, 29

[32] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible Query Processing in Starburst. In *SIGMOD*, pages 377–388, 1989. 19, 20, 47

[33] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD*, pages 287–298, 1999.

[34] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *SIGMOD*, pages 647–658, 2004. 7, 47

[35] Ihab F. Ilyas, Jun Rao, Guy Lohman, Dengfeng Gao, and Eileen Lin. Estimating Compilation Time of a Query Optimizer. In *SIGMOD*, pages 373–384, 2003.

[36] Yannis E. Ioannidis and Stavros Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, pages 268–277, 1991. 3, 32

[37] Navin Kabra and David J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *SIGMOD*, pages 106–117, 1998. 4, 14

[38] Per-Åke Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. Cardinality Estimation Using Sample Views with Quality Assurance. In *SIGMOD*, pages 175–186, 2007. 23

[39] Wolfgang Lehner, Roberta Cochrane, Hamid Pirahesh, and Markos Zaharioudakis. fAST Refresh using Mass Query Optimization. In *ICDE*, pages 391–398, 2001. 18

[40] Sam Lightstone, Guy M. Lohman, and Daniel C. Zilio. Toward Autonomic Computing with DB2 Universal Database. *SIGMOD Record*, 31(3):55–61, 2002.

[41] M. O. Lorenz. Methods of Measuring the Concentration of Wealth. *Publications of the American Statistical Association*, 1905.

[42] Volker Markl, Nimrod Megiddo, Marcel Kutsch, Tam Minh Tran, Peter J. Haas, and Utkarsh Srivastava. Consistently Estimating the Selectivity of Conjuncts of Predicates. In *VLDB*, pages 373–384, 2005. 8, 35, 87, 88, 89, 100, 102, 103, 106, 117

[43] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust Query Processing through Progressive Optimization. In *ACM SIGMOD*, pages 659–670, 2004. 4, 14

[44] Frank Olken and Doron Rotem. Simple Random Sampling from Relational Databases. In *VLDB*, pages 160–169, 1986. 25

[45] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *VLDB*, pages 486–495, 1997. 7

[46] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *SIGMOD*, pages 294–305, 1996. 47

[47] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[48] Claude E Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948. 29

[49] Utkarsh Srivastava, Peter J. Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. In *ICDE*, 2006. 14, 29, 44

[50] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001. 4, 13, 36

[51] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, pages 101–110, 2000. 17, 18, 62

[52] Xiaohui Yu, Nick Koudas, and Calisto Zuzarte. HASE: A Hybrid Approach to Selectivity Estimation for Conjunctive Predicates. In *EDBT*, pages 460–477, 2006. 47

[53] Xiaohui Yu, Calisto Zuzarte, and Kenneth C. Sevcik. Towards Estimating the Number of Distinct Value Combinations for a Set of Attributes. In *CIKM*, pages 656–663, 2005.

[54] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering Complex SQL Queries using Automatic Summary Tables. In *SIGMOD*, pages 105–116, 2000. 20, 27

[55] Qiang Zhu, Brian Dunkel, Wing Lau, Suyun Chen, and Berni Schiefer. Piggyback Statistics Collection for Query Optimization: Towards a Self-Maintaining Database Management System. *Comput. J.*, 47(2):221–244, 2004. 14

[56] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, pages 1087–1097, 2004. 17, 18, 62