Robustness in Automatic Physical Database Design

Kareem El Gebaly

David R. Cheriton School of Computer Science University of Waterloo

Technical Report CS-2007-29

Robustness in Automatic Physical Database Design

by

Kareem El Gebaly

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

©Kareem El Gebaly 2007

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Kareem El Gebaly

I further authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Kareem El Gebaly

Abstract

Automatic physical database design tools rely on "what-if" interfaces to the query optimizer to estimate the execution time of the training query workload under different candidate physical designs. The tools use these what-if interfaces to recommend physical designs that minimize the estimated execution time of the input training workload. Minimizing estimated execution time alone can lead to designs that are not robust to query optimizer errors and workload changes. In particular, if the optimizer makes errors in estimating the execution time of the workload queries, then the recommended physical design may actually degrade the performance of these queries. In this sense, the physical design is risky. Furthermore, if the production queries are slightly different from the training queries, the recommended physical design may not benefit them at all. In this sense, the physical design is not general. We define *Risk* and *Generality* as two new measures aimed at evaluating the robustness of a proposed physical database design, and we show how to extend the objective function being optimized by a generic physical design tool to take these measures into account. We have implemented a physical design advisor in PostgreSQL, and we use it to experimentally demonstrate the usefulness of our approach. We show that our two new metrics result in physical designs that are more robust, which means that the user can implement them with a higher degree of confidence. This is particularly important as we move towards truly zero-administration database systems in which there is not the possibility for a DBA to vet the recommendations of the physical design tool before applying them.

Acknowledgements

Thank God.

I would like to thank Yara Mazen, my wife, for her support, patience, and great food that helped me through the second year of my Master program.

I would also like to thank my sister and brother in law Zeina El Gebaly and Dr. Wael Bekheet for their support, accommodation during my first year in my Master program.

I would also like to thank my parents for their support throughout my entire life.

I would also like to thank Ashraf Aboulnaga my supervisor.

I would also like to thank Prof. Tamer Özsu and Prof. Ken Salem for being my thesis readers.

I would also like to thank the University of Waterloo for providing an educational and professional environment.

Contents

1 Introduction **1**

1.1 Problem Definition 2

1.2 Thesis Contribution 4

1.3 Thesis Organization 5

2 Background and Motivation **6**

2.1 The State of the Art in Index Tuning 7

2.1.1 Workloads 7

2.1.2 Automatic Database Physical Design 8

2.1.3 Index Tuning Problem 10

2.2 Motivation for Robustness in Physical Design 12

3 Risk **14**

3.1 Motivating Risk 14

3.2 Motivating Examples 16

3.3 Risk Metric 19
3.3.1 MAXE Cost 19
3.3.2 MAXE Gap 21
3.3.3 Risk Metric 21
3.4 Examples of Risk in Physical Design 22
3.4.1 Example 1: Bitmap Index Scan 23
3.4.2 Example 2: Index Scan 24
3.4.3 Example 3: Less Risky Designs 25
3.5 Changes to PostgreSQL to calculate MAXE cost 25
3.5.1 Hypothetical Indexes 25
3.5.2 Determining MAXE cost 26

4 Generality in Physical Design **34**

4.1 Provisioning for Query Workload-changes 34
4.2 Motivating Generality 36
4.3 Generality in Physical Design 39
4.3.1 Example 1: Importance of Leading Columns 40
4.3.2 Example 2: Leading Columns and Overtraining 42
4.4 Generality Metric 43

5 Multi-objective Design Advisor **46**

5.1 Combining Benefit, Risk and Generality 46

5.2 Robust Index Tuning Problem 47

5.3 Multi Objective Design Advisor-MODA 48

6 Experimental Evaluation 52

6.1 Experimental Setup 52

6.2 Evaluating Risk 53

6.3 Generality 55

7 Related Work 68

7.1 Physical Design Advisor 68

7.2 Robust Query Optimization 70

7.3 Workload Compression 71

7.4 Automatic Statistics Collection 72

8 Conclusion and Future Work 74

8.1 Future Work 76

8.1.1 More Types of Robustness 76

8.1.2 Interaction With Statistics 76

8.1.3 Robustness in Online Automatic Physical Database Design 77

8.1.4 When to Re-tune? 77

8.1.5 Materialized Views and Physical Data Layout 77

A Changes in PostgreSQL Optimizer 85

A.1 Modified Data Structures in postgres.h 85

A.2 List of Affected Header files 86

A.3 List of Files Changed in PostgreSQL 88

A.4 Calculating the MAXE Selectivity 90

A.5 Calculating the MAXE cost 92

B TPC-H Variant Workload 94

List of Tables

4.1 Leading prefixes 38

4.2 Leading prefixes for the alternate configuration 39

4.3 Estimated execution time of query Q_1 using different index column orders . 40

4.4 Estimated execution time of query Q_2 using different index column orders . 41

List of Figures

2.1 Generic architecture for index tuning 8

2.2 Typical tuning scenario 11

2.3 Online tuning scenario (no DBA) 11

3.1 Low Selectivity, Not Correlated (LN) 29

3.2 Low Selectivity, Correlated (LC) 29

3.3 High Selectivity, Not Correlated (HN) 29

3.4 High Selectivity, Correlated (HC) 29

3.5 Query names and predicates selectivity and correlation 29

3.6 The effect of correlation on the cost of the Bitmap Heap Scan operator. . . 30

3.7 The effect of correlation on the cost of the Bitmap Heap Scan operator. . . 30

3.8 PostgreSQL optimizer costing 31

3.9 MAXE costing 31

3.10 Query Q_2 32

3.11 Query Q_3 32

3.12 Execution time vs degree of correlation–Bitmap Index Scan operator . . . 32
3.13 Execution time vs degree of correlation–Index Scan operator 33
3.14 A less risky design 33
4.1 Workload G 38
4.2 Number of unique prefixes a configuration of size $3DB$ 44
6.1 Effects of risk on synthetic workload on the TPC-H database 54
6.2 Effect of risk on TPC-H benchmark queries 54
6.3 Number of unique prefixes in configuration of size $2DB$ 56
6.4 Number of unique prefixes in configuration of size $2.5DB$ 56
6.5 Number of unique prefixes in configuration of size $3.5DB$ 57
6.6 Number of unique prefixes in configuration of size $4DB$ 57
6.7 Number of unique prefixes in configuration of size $4.5DB$ 58
6.8 Number of unique prefixes in configuration of size $5DB$ 58
6.9 Effect of increasing size constraint on number of unique prefixes 60
6.10 Effect of increasing size constraint on generality metric 60
6.11 Effect of increasing size constraint on performance 61
6.12 Performance on training and test workloads for configuration size 1DB . . 63
6.13 Performance on training and test workloads for configuration size 1.5DB . 63
6.14 Performance on training and test workloads for configuration size 2DB . . 64
6.15 Performance on training and test workloads for configuration size $2.5DB$. 64

6.16 Performance on training and test workloads for configuration size $3DB$ . . 65

6.17 Performance on training and test workloads for configuration size $3.5DB$. 65

6.18 Performance on training and test workloads for configuration size 4DB . . 66

6.19 Performance on training and test workloads for configuration size $4.5DB$. 66

6.20 Performance on training and test workloads for configuration size $5DB$ . . 67

A.1 Original code to calculate the selectivity of a predicate list 91

A.2 Original code to calculate the selectivity of a predicate list on an index . . 91

A.3 Modified code to calculate the MAXE selectivity of a predicate list 91

A.4 Modified code to calculate the MAXE selectivity of a predicate list on an index 92

A.5 Modified code to propagate the MAXE cost 93

Chapter 1

Introduction

Pushing the complexity of concurrency issues, optimization, and transaction management to the database management system (DBMS) has led to a decrease in the cost of complex software systems. With information technology becoming, more and more, a part of our everyday life, online systems in areas as diverse as health care, banking, and government are collecting and processing large amounts of data around the clock, which is managed by DBMSes for use in on-line transaction processing (OLTP) or decision support system (DSS) applications. With the decrease in the cost of software and hardware, the cost of *database administration* is becoming more significant compared to the total cost of ownership (TCO) [32]. This makes it important to simplify the role of the database administrator (DBA) by making the task less demanding in terms of time, manpower, and expertise.

An important topic of research in the field of data management is self-managing databases, which aims at finding solutions towards alleviating the burden on the DBA or, more am-

bitiously, totally replacing the DBA. The goal is to alleviate some of the tasks that must be done by the DBA by providing a suite of tools that automates many of the necessary duties. One important class of these tools is the automatic physical database design tools that are an integral part of automated database tuning [23].

1.1 Problem Definition

Automatic physical database design tools rely on "what-if" interfaces to the query optimizer to estimate the execution time of the training query workload under different candidate physical designs. The tools use these what-if interfaces to recommend physical designs that minimize the estimated execution time of the input training workload. Minimizing estimated execution time alone can lead to designs that are not robust to query optimizer errors and workload changes. In particular, if the optimizer makes errors in estimating the execution time of the workload queries, then the recommended physical design may actually degrade performance. In this sense, the physical design is risky. Moreover, if the production queries are slightly different from the training queries, the recommended physical design may not benefit them at all. In this sense, the physical design is not general. We define *Risk* and *Generality* as two new measures to evaluate the robustness of a proposed physical database design, and we show how to extend the objective function being optimized by a generic physical design tool to take these measures into account. This is particularly important as we move towards truly zero-administration database systems

in which there is not the possibility for a DBA to vet the recommendations of the physical design tool before applying them. We have implemented a physical design advisor in PostgreSQL [1], and we have integrated our metrics into this design advisor.

All recent automatic physical design solutions that we are aware of rely on query optimizer costing to evaluate the benefit of candidate physical designs, augmenting the optimizer with "what-if" interfaces to create hypothetical physical designs. Design decisions such as choosing an index are based on query optimizer costing. Query optimization is far from being accurate or optimal. Mitigating query optimizing errors in the field of robust query optimization and execution has been tackled by various works [8, 9, 29]. However, there is no physical design advisor that is robust to query optimizer errors. One of the contributions of this thesis is that we characterize the effect of query optimizer costing problems in terms of physical design *risk*. We define a risk metric that quantifies the robustness of index configurations to optimization errors. Integrating the risk metric into the design advisor results in physical designs that are more robust.

Automatic physical design literature adopts a workload-based approach. In a workload-based approach the assumption is that the system is given a representative SQL query workload that resembles the future workload in the production environment. We refer to the workload given to the physical design advisor as the *training workload*. Physical design tuning tools try to find the physical design (indexes, materialized views, etc.) that minimizes the estimated runtime of the training workload. We note that if the production workload is even slightly different from the training workload, the production workload

may not benefit from the chosen physical design. The difference between the production and training workloads may be in query frequencies, parameter values, or attribute mixes, and any of these differences may prevent the production workload from benefiting from the chosen physical design. In this sense, the physical design is *overtrained* to the training workload and lacks *generality*. We define a metric that quantifies generality, and we integrate this generality metric into our design advisor to create physical designs that are more robust.

1.2 Thesis Contribution

- Introducing a risk metric that quantifies the robustness of a given physical design configuration to query optimizer errors.
- Introducing a generality metric that quantifies the robustness of a given physical design configuration to workload changes.
- Multi-objective Design Advisor (MODA): A novel technique for physical design tuning that finds more robust physical designs by finding physical configurations that maximize an objective function combining benefit, risk, and generality.
- Including DBA preferences through weights on the different metrics in MODA that allow the DBA to specify the relative importance of benefit in estimated execution time, risk, and generality.

1.3 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we give the background and problem definition, and we present the current state of the art in automatic physical design. In Chapter 3, we explain the motivation behind the risk metric and details about the sources of errors in query optimizers and their relevance to physical design. We also define our risk metric and show how to calculate it using Minimal Assumption Extreme cost Estimation (MAXE). We present the changes to the PostgreSQL DBMS required for MAXE. In Chapter 4, we present the problem of overtraining in physical design, and we define our generality metric. Chapter 5 describes the Multi-objective Design Advisor (MODA) that uses our metrics. In Chapter 6, we give an experimental evaluation of our approach. Chapter 7 presents related work. Conclusions and suggestion for future work are presented in Chapter 8.

Chapter 2

Background and Motivation

Choosing a good physical design (indexes, materialized views, vertical and horizontal partitioning, etc.) is an essential task for almost any database workload [23]. Physical design tuning has a great impact on the performance of database workloads, up to orders of magnitude. Choosing *index configurations* is an important part of physical design tuning, and is the focus of this thesis. It is a task that requires much expertise from the DBA and consumes time and effort. Automatic physical design tools help in this task and might ultimately replace the DBA. Currently, most commercial database management systems like IBM DB2 [33], Microsoft SQL Server [5], and Oracle [24] have built-in index tuning tools, or *physical design advisors*. These advisors choose a set of indexes that yields good performance for the given workloads.

2.1 The State of the Art in Index Tuning

2.1.1 Workloads

DBMS tools, such as Oracle's Automatic Workload Repository [24], collect workloads over long periods of time for use in database tuning. Statements collected and tuned are the **SQL DML** statements (**SELECT**, **INSERT**, **UPDATE**, and **DELETE**). Workloads are collected over a long period of time and are often too large and redundant. Large workloads hurt the performance of physical design tuning because the time needed to tune a workload is dependent on the number of statements in the workload. Thus, the collected SQL statements usually go through a phase of pruning referred to as *workload compression*. The goal of workload compression is to reduce the number of statements in the workload in order to save time in the tuning phase. The key idea of workload compression is to come up with a reduced set of statements that are *representative* of the whole workload. Another important challenge is that workload compression should not take longer than the benefit in the time compared to tuning the original workload. Current techniques for workload compression are presented in Chapter 7

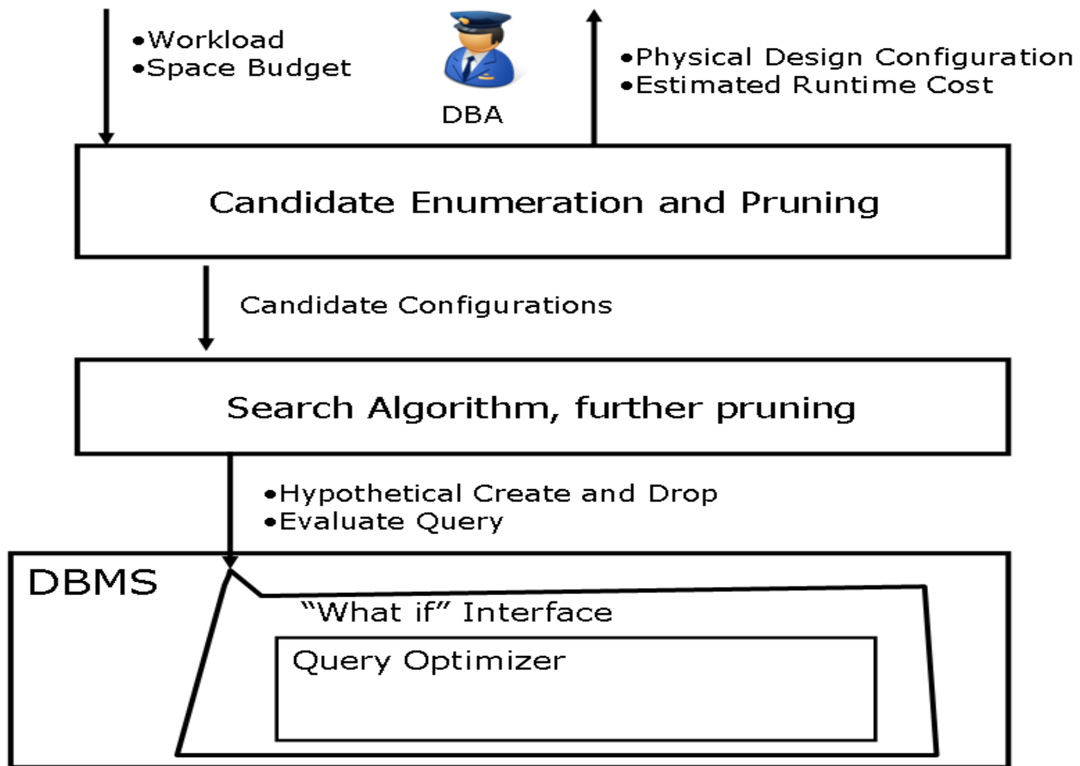

Figure 2.1: Generic architecture for index tuning

2.1.2 Automatic Database Physical Design

Figure 2.1 shows the architecture of a typical physical design advisor. The architecture is divided into three layers: candidate enumeration, the search algorithm, and the "what-if" interface inside the DBMS.

1. **Candidate Enumeration**

   Current design advisors use a cost-based approach for index tuning. The typical approach is that the DBA gives the design advisor a query workload and a space budget. The estimated cost of the workload is evaluated under various candidate

index configurations. Given a workload the physical design advisor starts by generating candidate indexes. These candidate indexes define the space of possible physical designs. The common strategy for choosing the candidate indexes is selecting the syntactically relevant indexes by analyzing the queries. For instance, columns mentioned in the **SELECT**, **WHERE**, and **ORDER BY** clauses are syntactically relevant. Columns in the **SELECT** clause may be included as an index suffix (values included in the leaf nodes of the B-tree only), or as index keys. Columns used in the **WHERE** clause are important because they specify which index keys are useful. Finally, columns mentioned in the **ORDER BY** clause can be included in index keys to save sorting time [20].

2. **Search Algorithms**

   Index tuning techniques need some search strategy to find the index configuration that minimizes the estimated cost of queries under space budget constraints. Enumerating all possible index configurations is not feasible because of the exponential number of possibilities. The problem is further complicated by the issue of *index interaction*: the benefit of an index in a configuration depends on other indexes in this configuration. Thus, current tools use approximate search techniques that use heuristics and greedy search [20, 31].

3. **What-if Interface**

   It is not feasible to materialize proposed physical designs and then execute queries

over the workload to evaluate the quality of these designs. Instead, query optimizer cost estimates are used to evaluate physical designs. Physical design advisors do not create indexes to evaluate them, since indexes are expensive to create and drop. They use a "what-if" mode of the query optimizer in which indexes are simulated by inserting metadata and statistics into the catalog. This approach was first introduced in [26] and is detailed in [19].

2.1.3 Index Tuning Problem

Current database physical design advisors only consider (a) storage size constraints and (b) benefit in query optimizer estimated workload running time. For large databases it is always important to impose a storage size constraint, which is the maximum allowable size for the proposed configuration, since the space required for useful indexes can be very large. Definition 1 formally defines the index tuning problem.

Definition 1 *Index Tuning Problem:*
Given a database D, a workload W consisting of a set of SQL statements, and a storage constraint S, find the set of indexes with total size less than S that minimizes the estimated runtime cost of W on D.

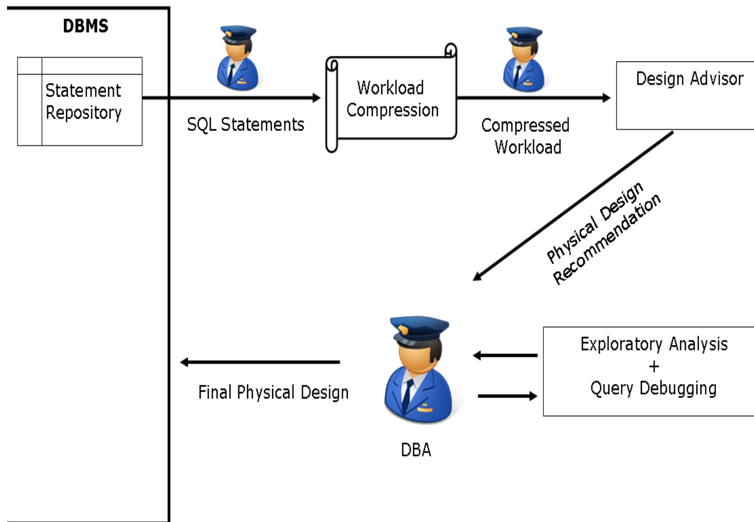

Figure 2.2: Typical tuning scenario

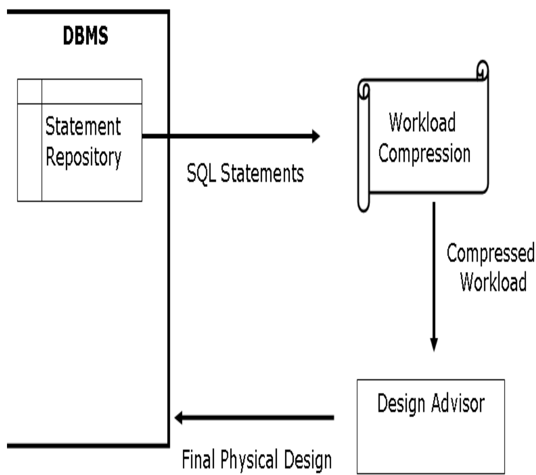

Figure 2.3: Online tuning scenario (no DBA)

2.2 Motivation for Robustness in Physical Design

Self-managing database literature provides a semi-automatic solution to the index tuning problem. Figure 2.2 depicts the typical physical design tuning scenario. During production runs, a stream of SQL statements is logged into the statement repository. Sometimes the DBA adds more queries from applications in the system (e.g. static prepared statements). The SQL statements are then collected and compressed by the DBA, sometimes with the help of a workload compression tool. This workload is given to the physical design advisor to tune along with a space budget. After tuning the workload using the physical design advisor the DBA takes the recommended physical design and performs further analysis. State of the art DBMSes are shipped with a suite of tools that help the DBA in this debugging and exploratory analysis [5]. Exploratory analysis is done via hypothetical physical design interfaces that answer "what-if" questions without actually building the access paths. This iterative process of debugging and analysis ends with choosing the final physical design.

The classical physical design tuning scenario assumes that the DBA is deeply involved in the process. Recently, there is a trend towards on-line tuning advisors [16], in which the DBA is more and more pushed outside of the loop. The desired approach is depicted in Figure 2.3, with no DBA involved in the process. Queries stream into an SQL statement repository and are analyzed by the physical design advisor. The physical design recommendations make their way directly to the production environment. It is imperative in such a scenario, that automatic physical designs become less workload sensitive and more

robust, hence more *admissible*. We argue that the current index tuning problem formulation is not the best in terms of robustness. We argue that even with using exhaustive cost-driven algorithms, we will not get robust configurations because of two main reasons: (a) estimated costs suffer from inaccuracies, and (b) the training workload may not be representative of the production workload.

We introduce two new metrics that quantify the robustness of physical designs: *Risk* and *Generality*, described in Chapters 3 and 4, respectively. Risk captures the estimated penalty of the worst case scenario if the optimizer assumptions fail. Generality reduces the effect of workload overtraining. Our approach does not ignore the estimated cost. Our new metrics define new dimensions for evaluating the quality of a physical design, while benefit in execution time based on optimizer estimates represents another dimension of quality. We combine these quality metrics in a weighted sum to evaluate the overall quality of a physical design. The weights are determined by the user, and they signify the relative importance of each dimension of quality.

Chapter 3

Risk

In this chapter we present the risk metric. This metric quantifies the sensitivity of the estimated benefit of an index configuration to query optimizer errors. It attempts to measure the effects of the assumptions made by the optimizer in costing the chosen set of indexes. Our goal in defining this metric is to choose physical designs with minimal difference between query optimizer costs and the worst case costs that may be encountered for workload queries.

3.1 Motivating Risk

Query optimizer costing is known for its flaws. To name a few sources for cost inaccuracies: (a) inaccurate cardinality estimation, (b) inaccurate I/O costing, (c) crude modeling of buffers sizes and contents, and (d) crude modeling of overheads due to concurrent access

and transaction management. Flaws in optimizer costing are a major reason for poor query performance. In most cases, query optimizers perform very well and come up with near optimal plans but in some cases they pick very bad plans, typically because they significantly underestimate the cost of the chosen plan.

Query optimizers rely mainly on database statistics to estimate cardinalities, which is essential for finding optimal plans and estimating their costs. Optimizer cost estimates are also used by the "what-if" interface that the physical design advisor uses. These cost estimates depend heavily on the accuracy of the query optimizer cost model and the data statistics. In general, single column statistics can safely be assumed to be fresh and accurate. They are cheap to collect and maintain using techniques like [2]. Techniques like [2, 17] can be used to collect and maintain multi-column statistics. For collecting statistics to estimate join cardinality, techniques like [10, 11, 12, 25] are used to collect statistics on intermediate query expressions. Due to the exponential blowup in the number of possible multi-column and join statistics, and to the complexity of collecting these statistics and representing multi-dimensional distributions, neither of these types of statistics can be safely assumed to be present in the system or accurate. Hence, query optimizers rely on assumptions about data distributions. One of the assumptions that is universally used and that significantly affects accuracy is the independence assumption, used to estimate cardinality for multi-column predicates. Query optimizers that use the independence assumption to evaluate the joint selectivity of a multi-column predicates uses the product of the selectivities of each predicate as an estimate of the joint selectivity.

These assumptions are a major cause of inaccuracies in cost estimation. If the physical design advisor bases its decisions on inaccurate costs, it may make poor choices. The more sensitive a physical design is to cost estimation inaccuracies, the more *risky* it is.

In general, DBAs could detect such inaccuracies using their expertise and domain knowledge. They would be able to reject risky indexes after doing some query debugging and profiling. However, since we are focusing on online physical tuning [16] with true zero database administration, there is no DBA to vet the decisions of the physical design tool. Thus it is important to automatically avoid risky configurations in the physical designs tools.

3.2 Motivating Examples

To further motivate risk, we focus on errors due to predicate correlation on the TPC-H Benchmark database with scale factor (1GB) [30]. For these and other examples in this thesis we are using PostgreSQL. The detailed setup for our experiments is given in Chapter 6. Consider the two columns **shipdate** and **receiptdate** in relation **LineItem**. A database administrator could deduce through domain knowledge that conjunctive predicates on columns **shipdate** and **receiptdate** with the same date range are highly correlated. We would like to examine the estimated cost and runtime cost of selective versus non-selective and correlated versus non-correlated predicates on these columns, with and without indexes.

We choose two time intervals: the *High Selectivity* time interval is 10 days and the *Low Selectivity* time interval is 1 month. We know that range predicates on **shipdate** and **receiptdate** are correlated if the time intervals for both predicates correspond exactly. On the other hand, if the time interval on **receiptdate** is before **shipdate** the predicates are negatively correlated. Therefore, we create the four queries shown in Figures 3.1-3.4.

In the four queries we **SELECT AVG** (**ExtendedPrice**) **FROM LineItem** where the predicates on columns **shipdate** and **receiptdate** are varied. We ran the queries in the indexed setup and non-indexed setup. In the indexed setup we create an index on **LineItem**(**shipdate**,**receiptdate**). Based on our knowledge of the query optimizer, we know that it will choose the index to execute the queries. In the non-indexed setup, we have no indexes on relation **LineItem**, forcing the query optimizer to choose a sequential scan on the relation. In this example, we would like to show two things: the effect of correlation on query optimizer costing, and its effect on actual cost.

Figure 3.6 shows the estimated cost of the four queries. In all cases the estimated cost using a sequential scan is significantly higher than the estimated cost using indexes. Figure 3.7 shows the actual runtime of the queries. The runtime of the uncorrelated queries is indeed significantly lower using an index versus using sequential scan. On the other hand, the runtime of the correlated queries using an index is *much worse* than the runtime using sequential scan. The query optimizer makes the error because it is assuming independence, so it underestimates the combined selectivity of the correlated predicates on shipdate and receiptdate. This error leads to the benefit of the (shipdate, receiptdate)

index being estimated to be much higher than it really is. Therefore, this index will most likely be selected over potentially more useful indexes or even a cheaper sequential scan. In this sense, the index is *risky*. The query optimizer and candidate enumeration approach of the design advisor combined have no means to avoid this. In fact, we argue that a good design advisor enumeration algorithm must choose indexes on selective predicates as candidate indexes.

Our goal in this chapter is to avoid risky indexes like this one. The classical approach to physical design tuning uses the query optimizer costing as the sole cost function. In this approach, the query optimizer is called repeatedly by the design advisor to estimate the benefit of using certain indexes. If the query optimizer overestimates the selectivity of some predicates it will report a lower estimated execution time to the design advisor. This will cause the design advisor to choose indexes based on wrong information, choosing less useful or even bad indexes over more useful ones. This has two undesirable effects. If the index is used, it will be involved in a wrong plan that will cause a performance degradation instead of benefit. At best, if the query optimizer is a learning or proactive optimizer the index may not be used in the future. In this case, the index will end up hurting the performance of update statements and wasting a useful part of the disk space budget.

3.3 Risk Metric

3.3.1 MAXE Cost

As we have seen in Section 3.2, there is a *risk* that in the worst case using an index performs worse than a sequential scan. Therefore, in our approach, we choose to provision not only for the expected case but also for the worst case. We assume that the query optimizer returns two costs: a *normal* cost and a *worst case* cost. The normal cost is the unchanged query optimizer estimated cost, while the worst case cost is calculated assuming the optimizer assumptions are violated in the worst possible way. In this thesis we concentrate on the independence assumption because as we have seen earlier it is the most relevant in terms of cardinality estimation so it is a dominating factor in determining the runtime of an index based plan. Thus, to determine worst case cost, whenever the optimizer needs to estimate the joint selectivity of multi-column predicates we assume full correlation instead of assuming independence. We use a pessimistic approach and assume the worst and instead of multiplying the selectivity we take the *minimum of the selectivities*. We call this worst case costing method Minimal Assumptions eXtreme cost Estimation- or MAXE cost for short.

To support MAXE cost estimation, we need to modify the query optimizer to return two costs for each operator: the normal cost and MAXE cost assuming worst case cardinality for multi-columns predicates. We have made these changes in the query optimizer of PostgreSQL and we describe them in Section 3.5.2. The required changes in any DBMS

would be similar.

As an example of MAXE cost estimation consider query Q_1 on relations $R(a, b, c)$ and $S(a, b, c)$:

SELECT AVG $(S.b)$ **FROM** R,S **WHERE** $R.a = S.a$ and $P_1(R.b)$ and $P_2(R.c)$;

P_1 and P_2 are predicates that could be equality, range, or other predicates. Figures 3.8 and 3.9 show the plan chosen by the PostgreSQL query optimizer annotated with normal and MAXE cost and cardinality, respectively. The optimizer estimated using its normal assumptions that the combined selectivity of P_1 and P_2 is very high, generating three rows. The cost of one sequential scan on relation R and three sequential scans on S is approximately $603, 070$. The cost is then propagated to the aggregate operator which has a negligible cost so the total cost of the plan is $603, 070$. The MAXE cost estimation relaxes the independence assumption used to calculate the combined selectivity of P_1 and P_2 and chooses the minimum selectivity of both predicates. Using the MAXE cost P_1 and P_2 are estimated to generate 3257 rows. Therefore, the MAXE cost of the plan is the cost of one sequential scan over relation R and 3257 sequential scans over relation S which is approximately $307, 691, 163$. The MAXE cost is then propagated to the aggregation operator. The cost of sorting 3257 rows is 8 and the cost of the MAXE cost of the plan is $307, 691, 171$.

3.3.2 MAXE Gap

MAXE cost depends on the chosen plan, which depends on the index configuration. We define the MAXE gap as the ratio between the MAXE cost of the workload on index configuration C and normal cost of the workload on configuration C.

$$MAXE\ Gap = \frac{MAXE(W,C)}{Cost(W,C)} \tag{3.1}$$

The primary goal of a physical design advisor is to find the set of indexes that minimizes the estimated runtime cost. The MAXE gap quantifies how sensitive a plan is to optimizer costing errors. This sensitivity depends on the available indexes in the configuration, and it can be used as a measure of the riskiness of the configuration. Thus, a secondary goal of the physical advisor could be minimizing the MAXE gap to reduce the riskiness of the chosen physical design.

For example, the MAXE gap of the plan chosen for query Q_1 is $(307,691,171)/(603,070)$ which is approximately 510. This means that in the worst case the query plan may be 510 times worse than expected.

3.3.3 Risk Metric

Directly minimizing the MAXE gap of the configuration chosen by the design advisor can yield poor results because it ignores the MAXE gap of the original configuration. To account for the query planner assumptions we take the summation of the inverse of the

MAXE gap of each query in the workload in the configuration multiplied by the initial MAXE gap to eliminate the effect of the initial MAXE gap. We normalize the risk metric by dividing this summation by the number of the queries in the workload and subtracting one. We end up with a number that quantifies the risk. The smaller the number the higher the risk.

Definition 2

$$risk(W, C_O, C_N) = \left(\frac{\sum_{q \in W}\left(\frac{Cost(q,C_N)}{MAXE(q,C_N)} * \frac{MAXE(q,C_O)}{Cost(q,C_O)}\right)}{|W|}\right) - 1$$

C_O is the initial default configuration and C_N is the configuration being evaluated.

3.4 Examples of Risk in Physical Design

To demonstrate risk in physical design we consider the query templates Q_2 and Q_3 on relation $R(a, b, c)$ shown in Figures 3.10 and 3.11, respectively. P_1, P_2, and P_3 are range predicates with varying selectivity. We create synthetic relations with three attributes (a, b, c). To be able to control the degree of correlation we generate the three columns a, b, and c as independent random variables. We then choose a percentage of the rows and make their a and b columns equal, thereby introducing correlation. In our examples we construct 11 relations with different degrees of correlation. All relation have 16 million randomly generated rows. For ease of exposition we follow an intuitive naming scheme for the relations. Relation R_0 has no correlation because we change 0% of its rows. Relation

R_10 will has 10% of the tuples with columns a and b correlated, and finally relation R_100 will have all its tuples a and b correlated. We generate uniformly distributed random variables in the range $[1 - 16 * 10^6]$ for attributes a, b, and c to generate the 11 relations.

3.4.1 Example 1: Bitmap Index Scan

In the first example we set the selectivity of predicates $P_1(a)$ and $P_2(b)$ to both be 0.1. Using the independence assumption, the query optimizer will estimate the selectivity of the **WHERE** clause in Q_2 to be 0.01. Therefore, an index on columns a, b would be a good candidate index for reducing the execution time of Q_2, assuming that tuples are not correlated. Figure 3.12 shows the effect of correlation on actual execution time of Q_2 using the Bitmap Index Scan relational operator on index $I(a, b)$ and the Sequential Scan operator. The x-axis shows the percent of correlated rows and the y-axis shows the actual execution time. At R_0, with 0% correlated rows, the independence assumption holds and the execution time of the bitmap index scan is a fraction of the execution time of the sequential scan. Therefore, the benefit in execution time of using this index is very high. The benefit goes down very fast as the degree of correlation increases, and execution time becomes worse than a sequential scan because the number of I/O requests increases. It is also useful to note that at higher degrees of correlation the execution time goes down and becomes similar to the cost of a sequential scan because file system pre-fetching detects the frequent I/O requests to the same file and pre-fetches the file containing the relation in a sequential fashion. The figure shows us that index $I(a, b)$ is risky for Q_2. The estimated

cost for query Q_2 using this index is $36,034$ while the MAXE cost is $118,839$, giving a MAXE Gap of 3.3. This MAXE Gap tells us that in some situations the cost of a plan using this index may be 3.3 times worse than estimated. Thus we can see from the MAXE Gap that a physical design that has this index is risky.

3.4.2 Example 2: Index Scan

In this example we set the selectivity of predicates $P_1(a)$ and $P_2(b)$ to 0.001 and 0.001 respectively. Using the independence assumption, the query optimizer will estimate the selectivity of the **WHERE** clause as 0.00001. Therefore, an index on columns a, b would be a good candidate index for reducing the execution time of Q_2, but PostgreSQL would choose an Index Scan operator not a bitmap index scan operator. In this example we show the effect of correlation on the runtime performance of this index scan operator. The index scan operator does not sort the *record ids (rids)* retrieved from the B-tree before accessing the heap file. It is chosen by the query optimizer for highly selective predicates. Figure 3.13 shows the effect of correlation on the execution time of query Q_2 using index scan on $I(a, b)$ and sequential scan. The estimated benefit of using the index scan operator compared to using sequential scan is significantly high. But as shown in the figure the actual runtime is highly sensitive to the percentage of correlated rows. Here the estimated runtime cost for Q_2 is 301 while the MAXE cost is $29,410$ giving a MAXE Gap of 97.7, illustrating the riskiness of this index configuration.

3.4.3 Example 3: Less Risky Designs

In this example we set the selectivity of predicates $P_1(a)$, $P_2(b)$ and $P_3(c)$ to 0.001, 0.001, and 0.01, respectively. We restrict our choice of index configurations to one index. We either choose index $I(a,b)$ or $I(c)$. Given $I(a,b)$ the query optimizer will choose to use an index scan for query Q_2 and sequential scan for Q_3. Given $I(c)$ the query optimizer will choose a sequential scan for Q_2 and bitmap index scan for Q_3. Figure 3.14 reports the runtime of both queries, once given $I(a,b)$ or $I(c)$. The runtime of both queries using $I(c)$ is less sensitive to the degree of correlation. Although the estimated benefit of using $I(a,b)$ is higher than using $I(c)$, using index $I(a,b)$ is more risky.

3.5 Changes to PostgreSQL to calculate MAXE cost

3.5.1 Hypothetical Indexes

We have made changes to PostgreSQL 8.1 to add the server side extensions required for our work. We added a new keyword **HYPOTHETICAL**. Starting a CREATE INDEX command with HYPOTHETICAL will create the index as a hypothetical index. The PostgreSQL server is allowed to create the index in the catalog but it is preempted before actually building the index. The index size is an important statistic, so we make sure it is computed correctly. PostgreSQL does not consult the catalog when asking for the index size, but rather consults the PostgreSQL storage layer that probes the file system to get the file size of the index. The necessary changes have been implemented so that the routine

responsible for this file system probe will return the desired index size of the simulated index.

3.5.2 Determining MAXE cost

To estimate worst case selectivity, we need to modify two routines:

1. Clause List Selectivity Estimation:

   This routine is used by the PostgreSQL query optimizer to compute the selectivity of conjunctive and disjunctive lists of predicates on the same relation. Predicates may be any of the following: equality predicates, inequality predicates, or expression clauses (e.g. like predicates). In this selectivity estimation the optimizer reduces redundant and overlapping predicates. It then computes the isolated selectivity of every predicate. The basic assumption used to evaluate the combined selectivity of the list of predicates is that predicates are independent, so the optimizer multiplies the selectivities of the predicates to evaluate the overall selectivity. This selectivity is used to estimate the normal (non-MAXE) plan cost.

   To compute the MAXE selectivity we select the minimum selectivity of all the predicates, which is the worst case selectivity in the case where all predicates are fully correlated, and we use this minimum selectivity as the overall selectivity of the predicate list. This MAXE selectivity is then used by the optimizer to estimate MAXE cost for the query plan.

2. Cost Bitmap And Node:

   This routine estimates the overhead of using an index to access a relation. It calculates the cost of retrieving rids or values from the B-tree index and not the cost of accessing the heap file containing the base relation. In this case the query predicates that match the index will be used to retrieve rids or values from the index. The cost of accessing the B-tree depends on the combined selectivity of the predicates matching the index. PostgreSQL uses the independence assumption to evaluate the combined selectivity.

   For the MAXE selectivity, we use, as before, the minimum selectivity of all the predicates (the worst case selectivity in the case where all predicates are fully correlated). By computing the selectivity of the predicates the number of matching B-tree leaves could then be estimated. The number of leaves is then used to estimate the number of sequential and random I/Os needed to access the index. The cost is then propagated to the calling routine to estimate the total plan cost.

After estimating a worst case selectivity and cardinality we also need to change the cost estimation functions of PostgreSQL to return two costs instead of one: the original query optimizer costing, and MAXE cost. A fundamental change that enables this computation, is changing the Selectivity, Cost, Row, and Group types in PostgreSQL from the **double** type to a data structure with two members that are double, one for the normal cost that is used for query optimization and one for the MAXE cost that is only used by the design

advisor. These modifications to PostgreSQL required changes in 35 files and involved 2300 lines of code. More details about the changes are given in Appendix A.

```
SELECT AVG (ExtendedPrice) FROM LineItem WHERE
shipdate BETWEEN '1997-01-28' and '1997-02-28'
AND receiptdate BETWEEN '1997-01-01' and '1997-02-01'
```

Figure 3.1: Low Selectivity, Not Correlated (LN)

```
SELECT AVG (ExtendedPrice) FROM LineItem WHERE
shipdate BETWEEN '1997-01-02' and '1997-01-10'
AND receiptdate BETWEEN '1997-01-01' and '1997-01-09'
```

Figure 3.2: Low Selectivity, Correlated (LC)

```
SELECT AVG (ExtendedPrice) FROM LineItem WHERE
shipdate BETWEEN '1997-01-09' and '1997-01-18'
AND receiptdate BETWEEN '1997-01-01' and '1997-01-09'
```

Figure 3.3: High Selectivity, Not Correlated (HN)

```
SELECT AVG (ExtendedPrice) FROM LineItem WHERE
shipdate BETWEEN '1997-01-04' and '1997-02-04'
AND receiptdate BETWEEN '1997-01-01' and '1997-02-01'
```

Figure 3.4: High Selectivity, Correlated (HC)

Predicate selectivity	Correlation	Query Name	Figure
Low	No	LN	3.1
Low	Correlated	LC	3.2
High	No	HN	3.3
High	Correlated	HC	3.4

Figure 3.5: Query names and predicates selectivity and correlation

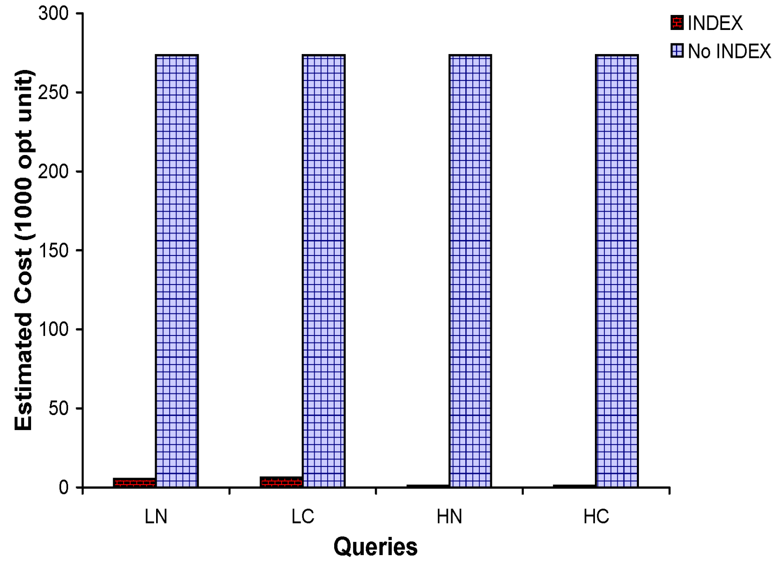

Figure 3.6: The effect of correlation on the cost of the Bitmap Heap Scan operator.

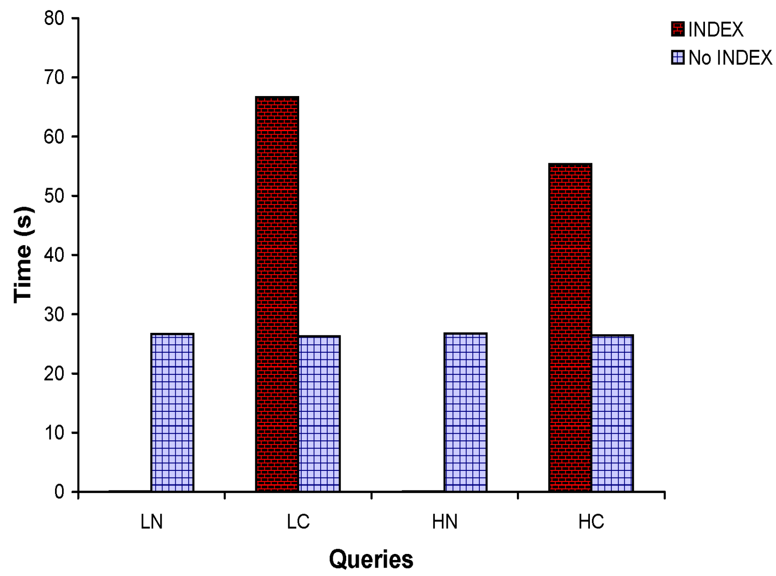

Figure 3.7: The effect of correlation on the cost of the Bitmap Heap Scan operator.

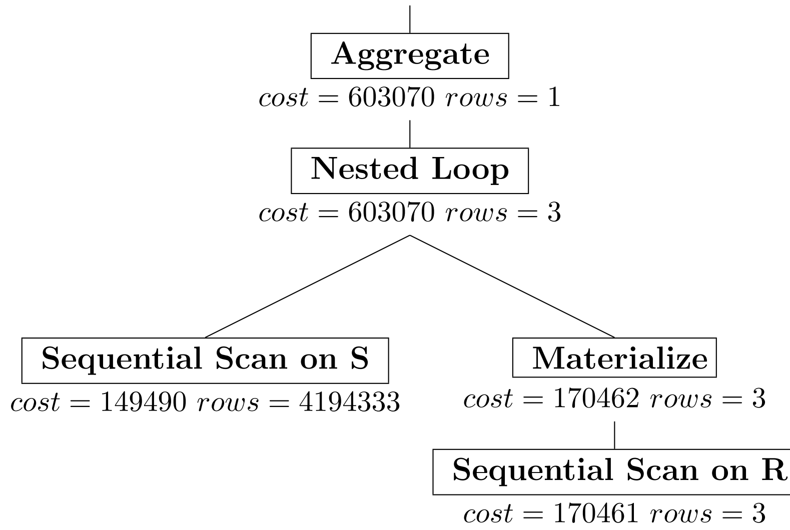

Figure 3.8: PostgreSQL optimizer costing

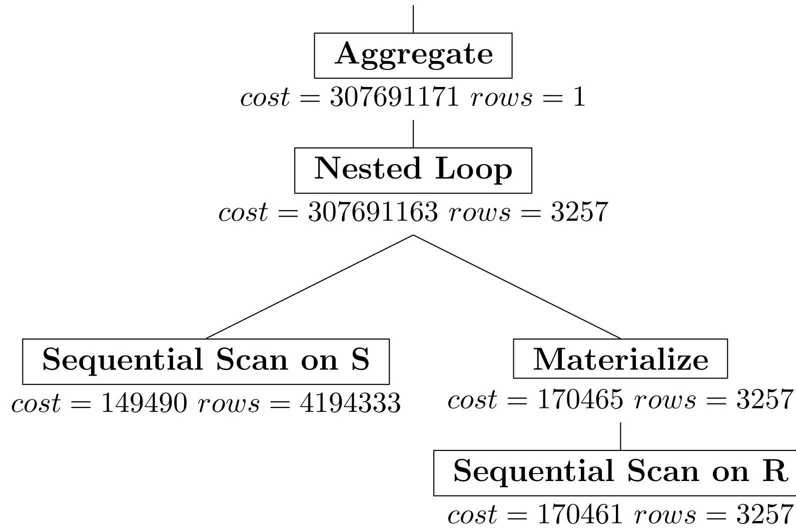

Figure 3.9: MAXE costing

SELECT AVG (c)
FROM R
WHERE $P_1(a)$ **and** $P_2(b)$

Figure 3.10: Query Q_2

SELECT AVG (a)
FROM R
WHERE $P_3(c)$

Figure 3.11: Query Q_3

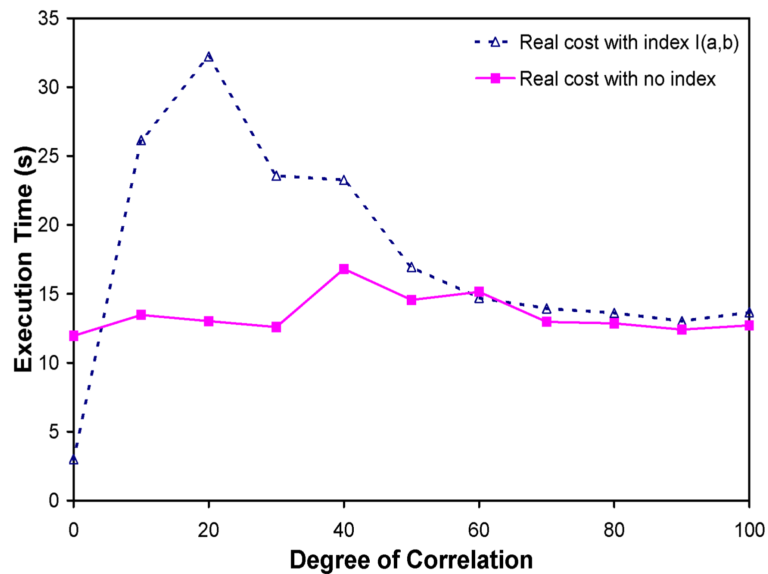

Figure 3.12: Execution time vs degree of correlation–Bitmap Index Scan operator

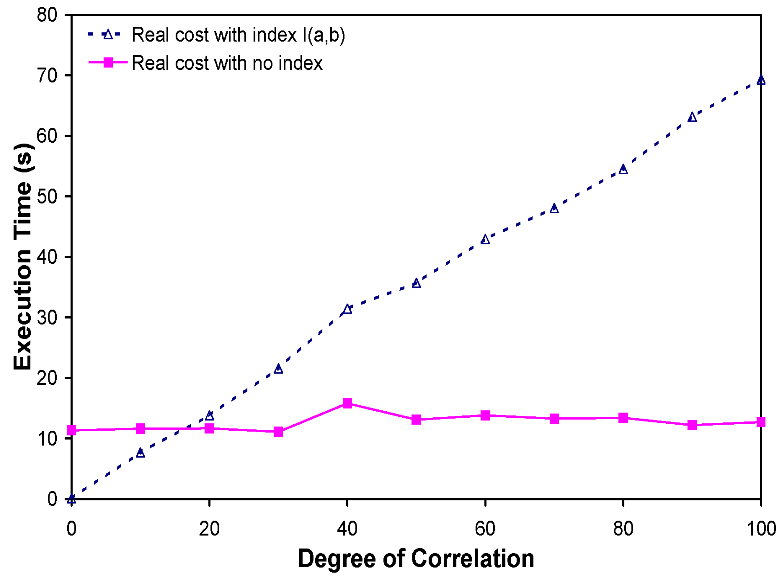

Figure 3.13: Execution time vs degree of correlation–Index Scan operator

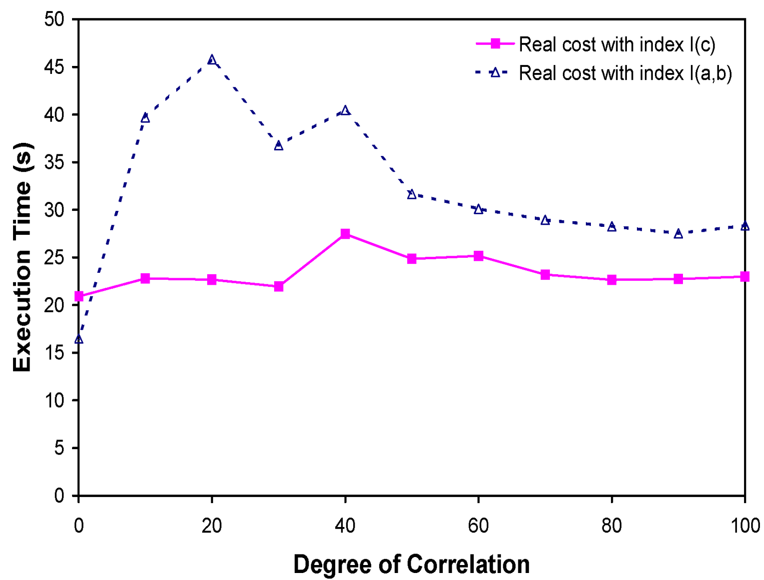

Figure 3.14: A less risky design

Chapter 4

Generality in Physical Design

4.1 Provisioning for Query Workload-changes

State of the art physical design advisors adopt a cost-based workload-aware approach. This approach assumes that the production workload will not be different from the training workload. As discussed in Section 2.1.1, the training workload is collected from past queries and prepared statements. Therefore, it is possible for the production workload to be different from the training workload, and it would be desirable for the design advisor to be able to deal with this possibility. Queries in the production workload could be any list of valid SQL queries, and they could be different from the training workload in parameter markers, frequencies, or query templates. We discuss each of these categories of differences next:

1. **Parameter Marker Change:**

   For this type of change query templates and frequencies in the production workload are the same as the training workload but the parameter marker values are different. This case is relevant if the main source of query templates is prepared statements or precompiled applications. In such environments, the query frequencies and the query templates do not change. On the other hand, the values bound to the parameter markers in future queries will vary depending on the behavior of the application. Changes in parameter marker values reflect on the desired indexes in two ways. First, they may make indexes on new candidate column combinations desirable. Second, they may affect the desired index column order.

   We observe that it is possible to effectively provision for changes in parameter marker values by choosing index configurations that have a *higher number of unique index prefixes*. An index configuration with a higher number of unique prefixes will benefit a wider class of queries, so we say it is *more general*. We propose a generality metric that assesses how general the index configuration is along this dimension. Our metric quantifies how good an index configuration is in terms of the number of unique (non-redundant) prefixes. The less redundant the prefixes are, the better the generality of the index configuration.

2. **Query Frequency Change:**

   For this type of change, the query template and the parameter markers do not change

but query frequencies change. This case is relevant when the mix of application requests in the production workload is expected to differ from the training workload.

We note that there is room for provisioning for query frequency changes. The design advisor could recommend an index configuration that benefits many queries although it is not optimal. A candidate metric that can be used to evaluate how much an index configuration benefits all the different queries in a workload is the standard deviation of benefit across query templates. Although this metric may be used to provision for query frequency changes, we have not explored it further in this thesis.

3. **Query Template Change:**

   If query templates are not fixed, the cost-based workload aware approach is not fit for index tuning. There is no room for provisioning since the production workload may be arbitrarily different from any training workload. The best solution for index tuning in this case is to rely on heuristic-based physical designs.

4.2 Motivating Generality

In this section, we identify the problem of index advisor overtraining. The index advisor recommends physical designs that perform well for the training workload. However, these designs may not be optimal if the workload changes. Specifically, we focus on a workload with changed parameter marker values.

Let us consider the following workload and database. The database has one relation,

R with five attributes (a, b, c, d, x), each of type decimal. In this synthetic relation, rows are generated according to a uniform distribution in the range $[0 - 1]$. The attributes are independent, so query optimizer estimates are accurate. This is appropriate for our goal in this chapter, which is to provision for changes in the workload, not for optimizer errors. We have generated a sample R with 4M rows (size on disk 455 MB). We consider a workload G with four queries, given in Figure 4.1, that has an estimated runtime of 506K optimizer units on a configuration with no indexes. Let us assume that we have enough space for building any required indexes. Let us also assume that we are seeking the optimal estimated runtime configuration. The optimal configuration has four indexes $\{I(R.a)$, $I(R.a, R.b)$, $I(R.a, R.b, R.c)$, $I(R.a, R.b, R.c, R.d)\}$. In this optimal configuration, the index columns exactly match the query predicates. First, the column combination exactly matches the query predicates. Second, the column order in the index matches the selectivity of the query predicates, which minimizes the cost of the index scan. From a cost-based perspective this configuration is optimal as it gives the lowest estimated runtime, which is 311K optimizer units. However, it has two main drawbacks. First, the configuration is overtrained in the sense that it does not provision for parameter marker changes in workload queries. Second, it seems counter intuitive to build four redundant access paths to achieve the optimal estimated runtime. The index configuration has only 4 unique leading prefixes which are listed in Table 4.1. On the other hand, configuration $\{I(R.a)$, $I(R.b, R.a)$, $I(R.c, R.a, R.b)$, $I(R.d, R.a, R.b, R.c)\}$ would give a suboptimal estimated cost of 370K optimizer units, but the number of leading columns in this configuration is 10, listed in Table 4.2. Therefore,

SELECT AVG (x)
FROM R
WHERE a<0.01

SELECT AVG (x)
FROM R
WHERE a < 0.09 and b < 0.11

SELECT AVG (x)
FROM R
WHERE a<0.25 and b< 0.3 and c< 0.35

SELECT AVG (x)
FROM R
WHERE a< 0.3 and b< 0.35 and c< 0.4 and d< 0.45

Figure 4.1: Workload G

Indexes	Leading Prefixes
$R(a)$	$\{R(a)\}$
$R(a,b)$	$\{R(a), R(a,b)\}$
$R(a,b,c)$	$\{R(a), R(a,b), R(a,b,c)\}$
$R(a,b,c,d)$	$\{R(a), R(a,b), R(a,b,c), R(a,b,c,d)\}$
$\{R(a), R(a,b), R(a,b,c), R(a,b,c,d)\}$	$\{R(a), R(a,b), R(a,b,c), R(a,b,c,d)\}$

Table 4.1: Leading prefixes

it benefits a wider class of queries.

In general, the overtraining phenomena happens if the design advisor is allowed a relatively large disk space budget for the given workload. In this case, the design advisor will choose very specific indexes for every query. These specific indexes are highly likely to have many common prefixes. On the other hand, with a minimal sacrifice in the estimated runtime, a more general index configuration may be chosen. Therefore, we consider such designs naive and overtrained. An expert DBA would never allow these indexes to reach production unless she has prior knowledge that the index production workload will exactly match the training workload.

Indexes	Leading Prefixes
$R(a)$	$\{R(a)\}$
$R(b,a)$	$\{R(b), R(b,a)\}$
$R(c,a,b)$	$\{R(c), R(c,a), R(c,a,b)\}$
$R(d,a,b,c)$	$\{R(d), R(d,a), R(d,a,b), R(d,a,b,c)\}$
$\{R(a), R(b,a), R(c,a,b), R(d,a,b,c)\}$	$\{R(a), R(b), R(c), R(d), R(b,a), R(c,a),$ $R(c,a,b), R(d,a), R(d,a,b), R(d,a,b,c)\}$

Table 4.2: Leading prefixes for the alternate configuration

4.3 Generality in Physical Design

With some provisioning, the design advisor may recommend configurations that are general enough to accommodate changes in parameter markers and query frequencies. Intuitively, if the design advisor maximizes the number of prefixes it will be able to benefit a wider class of queries on the same columns. With a slight reduction in performance the design advisor may provision for a much wider class of workloads and not overtrain for the given training workload. We have observed that overtrained configurations have redundant indexes that have a lot of common prefixes.

Our solution is to prevent the index advisor from overtraining by penalizing redundancy in configurations. Before presenting our metric, we present examples on relation R, described above, to illustrate the importance of the number of index prefixes, and how an index advisor will react to an increase in the number of queries in the training workload, further supporting the goal of increasing the number of leading columns. Our examples will also illustrate the significance of the effect of changes in parameter markers on changes in the estimated query runtime. In these examples, as in the rest of this chapter, we use

Index Column Order	Estimated Runtime
∅	180965.54
a,b,c	40158.57
a,c,b	40158.57
b,a,c	44427.38
b,c,a	44427.38
c,a,b	48077.71
c,b,a	48077.71

Table 4.3: Estimated execution time of query Q_1 using different index column orders

query optimizer cost estimates since they capture the effects of overtraining without relying on the actual runtime.

4.3.1 Example 1: Importance of Leading Columns

Query predicates do not have to exactly match index columns. An index $I(a, b, c)$ could be used to reduce runtime of queries on columns $\{a, b, d\}$. In general, an index on columns $\{c_1, c_2, \ldots, c_n\}$ may replace an index on columns $\{c_1, c_2, \ldots, c_m\}$ where $n > m$ with minimal decrease in performance.

For example, consider the following query, Q_1:

SELECT AVG (x)

FROM R

WHERE $a < 0.1$ and $b < 0.125$ and $c < 0.15$

As shown in Table 4.3, the index column order that matches the predicate selectivities is the one that provides the minimum estimated execution time. We also note that the

Index Column Order	Estimated Runtime
$\emptyset$	180952.82
a,b,c	8899.77
a,c,b	8899.77
b,a,c	24767.27
b,c,a	24767.27
c,a,b	98867.91
c,b,a	98867.91

Table 4.4: Estimated execution time of query Q_2 using different index column orders

actual execution time follows a similar pattern. The key point we illustrate in Table 4.3 is that the column order does not affect the estimated execution time by much. For example, the index that yields the worst estimated execution time, namely index $I(c, b, a)$ (the one with reversed column order), is only 19.7% worse than the best index.

On the other hand, there are queries where index column order is very important. For example, consider the following query Q_2:

SELECT AVG (x)

FROM R

WHERE $a < 0.01$ and $b < 0.1$ and $c < 0.5$

As shown in Table 4.4 the index column order has a significant effect on performance in this case. Thus, it is important for the index advisor to include as many column orders as possible, potentially choosing column orders that are suboptimal where it does not matter (e.g. for Q_1).

4.3.2 Example 2: Leading Columns and Overtraining

In this scenario we create a synthetic relation called DB_{11C}. Relation DB_{11C} has 11 decimal columns each generated independently according to a uniform distribution in the range $[0 - 1]$. The relation has 4M rows. We generate a workload that contains 100 aggregate queries of the form: **SELECT AVG**(x) **FROM** R **WHERE** P-list

P-list is a list of conjunctive range predicates generated using the following random process. We first choose a random integer in the range $[1, 5]$ which determines the number of range predicates to use. We choose the columns to use with each predicate at random according to a uniform distribution. To determine the selectivity of each predicate we generate a uniform random variable in the range $[1, 10]$ to determine the selectivity class. We have chosen 10 different selectivity classes ranging from very high selectivity to very low selectivity. The predicates in the different selectivity classes have selectivities , $\{1 * 10^{-3}$, $5 * 10^{-3}$, $1 * 10^{-2}$, $5 * 10^{-2}$, $1 * 10^{-1}$, 0.5, 0.6, 0.7, 0.8, 0.9$\}$.

We have created three different workloads, which we call the left, right, and uniform workloads. To generate the left workload we restrict predicates on the first four columns to the five most selective predicate classes. To generate the right workload, we restrict predicates on the last four columns to the five most selective predicate classes. For the uniform workload there are no restrictions on the selectivity classes. We first choose the number of columns that will be used to generate the query. We then choose which columns to use. This workload will also be used in our experiments in Chapter 6. In this example we run a design advisor that resembles state of the art design advisors in that it chooses

configurations based only on benefit to queries. The design advisor is given a space budget equal to three times the size of the relation. We vary the number of queries that we use to train the design advisor, choosing from 100 queries in the left workload described above. Figure 4.2 shows the number of unique prefixes in the design recommendation versus the number of queries used as a training workload. As we can see, the number of unique prefixes increases by increasing the number of queries in the workload, which means that the design advisor becomes more general and less overtrained as it sees more queries. The design advisor that saw, say, 20 queries will not be able to benefit the remaining 80 queries in the workload. Our goal is to avoid this problem and make the design advisor choose configurations that can benefit queries beyond those seen in the training workload. We want the design advisor to choose more indexes but we would also like these indexes to be diverse and not just have redundant prefixes.

4.4 Generality Metric

We introduce a generality metric that assesses the quality of an index configuration in terms of overtraining. We quantify the redundancy in an index configuration by analyzing the number of unique leading prefixes in this configuration. A redundant index configuration suffers from a lower number of index prefixes, which determines the class of query predicates that can be served by the indexes in the configuration. The generality metric gives a number from $[0 - 1]$, where 0 means that the given configuration has very low generality while 1

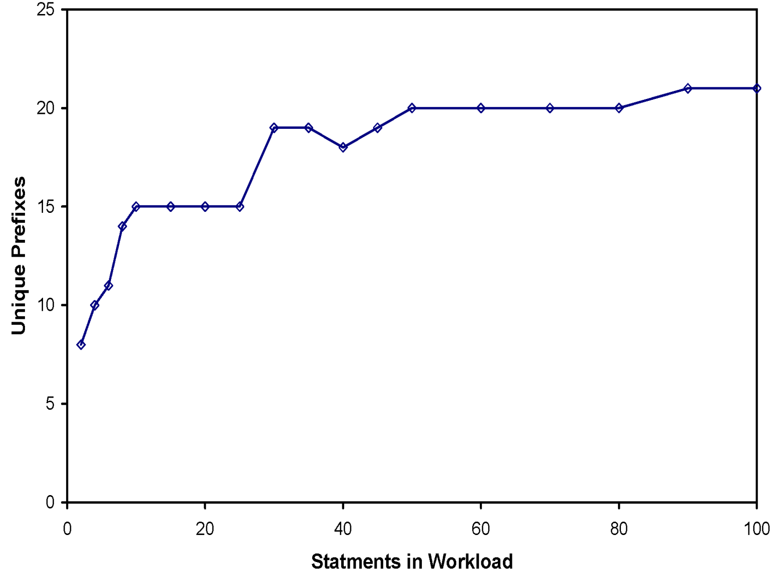

Figure 4.2: Number of unique prefixes a configuration of size $3DB$

means that the index configuration is the most general and has no redundant prefixes. The generality metric is defined as follows:

$$Generality(C_N) = \frac{Number\ of\ Unique\ Index\ Prefixes}{Maximum\ Number\ of\ Possible\ Unique\ Index\ Prefixes}$$

The number of unique index prefixes is the cardinality of the set containing all the leading prefixes in the configuration. The maximum number of possible index prefixes is calculated by adding the number of columns in every index. For example, the configuration $\{I(a,b,c),\ I(b,d)\}$ has two indexes: a three column index and a two column indexes. Index $I(a,b,c)$ may replace the three indexes $\{I(a), I(a,b), I(a,b,c)\}$ therefore the maximum

possible number of index prefixes associated to $I(a, b, c)$ is three. Index $I(b, d)$ may replace indexes $\{I(b), I(b, d)\}$ therefore the maximum number of prefixes associated with $I(b, d)$ is two. Therefore, the maximum number of possible index prefixes is five.

As we show later, this metric can be used to generate more general index configurations with minimal sacrifice in estimated runtime. This has great benefit if the workload changes, and it does not result in a big penalty when the workload does not change.

Chapter 5

Multi-objective Design Advisor

In this chapter we present a design advisor that proposes robust physical designs. Unlike current design advisors that focus solely on maximizing runtime benefit, our robust index tuning advisor has an objective function that combines benefit, risk, and generality.

5.1 Combining Benefit, Risk and Generality

In Chapters 3 and 4, we have introduced metrics for risk and generality. These metrics quantify the quality of a physical design along two new quality dimensions that can be used in addition to runtime benefit. We now introduce a weighted multi-objective cost function ρ that combines the benefit in estimated cost, risk and generality quality dimensions into one overall cost.

$$\rho(W, C_O, C_N, q) = q_1 * Benefit(W, C_O, C_N) + q_2 * Risk(W, C_O, C_N) + q_3 * Generality(C_N)$$

ρ takes as input an SQL query workload W which is the training workload. It takes two physical design configurations C_O and C_N. C_O is the initial (or default) configuration that we are trying to improve, and C_N is the index configuration to be evaluated. C_O is used as a reference configuration when calculating the benefit and risk of configuration C_N. q is a vector that holds three user defined weights that capture the relative importance of the different metrics. Risk and Generality are the metrics introduced earlier, and Benefit is defined as:

$$Benefit(W, C, C_O) = \frac{cost(W, C_O) - cost(W, C)}{cost(W, C)}$$

5.2 Robust Index Tuning Problem

The robust index tuning problem, which we aim to solve, differs from the traditional index tuning problem in that the goal is to maximize the objective function ρ, defined above. This simple change in the problem allows robust index tuning advisor to consider (a) storage size constraints and (b) benefit in query optimizer estimated workload running time but also take into account (c) provisioning for query optimizers errors and (d) query workload changes. Definition 3 formally defines the robust index tuning problem.

Definition 3 *The Robust Index Tuning Problem:*
Given a database D, a workload W consisting of a set of SQL statements, a storage constraint S, and a vector of weights on the different quality metrics q, find the set of indexes that maximizes $\rho(W, C_O, C_N, q)$, with total size less than S.

That is, we want to solve:

$$\arg\max_{C_N} \ \rho(W, C_O, C_N, q)$$

subject to: $size(C_N) < S$

5.3 Multi Objective Design Advisor-MODA

We have implemented a design advisor that optimizes the multi-objective cost function described above and so finds index configurations that solve the robust index tuning problem. We call our design advisor the Multi Objective Design Advisor (or MODA). If the user chooses to set the weights on risk and generality to zero, MODA will behave similarly to a normal index tuning advisor. On the other hand, if the user increases the weight on risk and generality, MODA will be biased towards choosing designs that provision for query optimizer errors and workload changes.

Algorithm 1 shows the behavior of our MODA system. In this thesis we want to examine the effects of the two proposed metrics–risk and generality–on the behavior of a design advisor. Hence, to focus on these two metrics, we have chosen to create a comprehensive

index tuning advisor that is guaranteed to choose index configurations that are similar to or better than state of the art index tuning advisors. State of the art tuning advisors use many techniques that significantly reduce the tuning time but that may reduce the quality of the chosen solution. First, index advisors ignore index interaction, meaning that the benefit of using an index is computed once compared to the default configuration [20, 31]. This is not accurate since the benefit of an index changes depending on what other indexes are available. Second, index advisors use further syntactic pruning techniques [21] that reduce the size of the search space but may prune away the optimal solution. Our algorithm does not do any pruning that would affect the quality of chosen the index configuration. We only prune syntactically irrelevant index columns and indexes that do not benefit any query in the workload. We also re-evaluate the benefit of all candidate indexes after choosing each index to take index interaction into account. Our algorithm only considers indexes on up to three columns to reduce the number of candidate indexes.

Algorithm 1 presents the outline of the MODA index recommendation process. In line 1 We start by generating candidate indexes by syntactically analyzing the workload. We consider any column mentioned in the **SELECT**, **WHERE**, **ORDER BY**, and **GROUP BY** clauses as a candidate index column. We also consider all combinations of two or three candidate columns as candidate indexes. Then, in lines $[2-4]$ we evaluate the current workload on the default configuration to get the initial estimated cost and MAXE cost of the workload. Next, in lines $[7-11]$ we use the "what-if" interface to evaluate the benefit and risk of every index. We also evaluate the generality of the configuration

given each index. Then in lines $[19-26]$ we take the best candidate index that fits the size budget and add it to the chosen set. To include index interaction in our future calculations we hypothetically create the indexes in the chosen set then we re-evaluate all the indexes. If an index gives a zero benefit we remove this index from the candidate indexes so it does not get re-evaluated in future iterations. We stop when there is no index with benefit in the candidate indexes that fits our size budget.

We have implemented the server side extensions required for MODA in PostgreSQL, as described in Chapter 3. The client side MODA application connects to this modified server and recommends index configurations for a given workload.

Algorithm 1 Multi-Objective Design Advisor - (MODA)

MODA(W,S,C_0,q)

1 $cands \leftarrow GenerateCands(W)$
2 $C_N \leftarrow C_0$
3 $cost_0 \leftarrow cost(W, C_0)$
4 $MAXE_0 \leftarrow MAXE(W, C_0)$
5 **for** $i \leftarrow 0$ **to** $|cands|$
6   **do**
7   Create Hypothetical Index($cand[i]$)
8         $cand[i].benefit \leftarrow benefit(cost_0, cost(W))$
9         $cand[i].risk \leftarrow risk(W, cost_0, MAXE_0, C_N)$
10         $cand[i].generality \leftarrow Generality(conf \cup cand[i])$
11         $cand[i].MODACOST \leftarrow q_1 * cand[i].benefit + q_2 * cand[i].risk + q_3 * cand[i].generality$
12   **if** ($cand[i].benefit <= 0$)
13   **then**
14   remove $cands[i]$ from $cands$
15
16   Drop_Hypothetical_Index($cand[i]$)
17
18 Sort $cands$ on $MODACOST$
19 **if** there is an index in $cands$ with size $< S$
20   **then**
21         $conf \leftarrow conf \cup cand[i]$
22         $S \leftarrow S - cand[i].size$
23   Create Hypothetical Index($cand[i]$)
24   **goto** 5
25   **else**
26   **return** $conf$

Chapter 6

Experimental Evaluation

6.1 Experimental Setup

In our experiments we use PostgreSQL server modified as described in Section 3.5. The client side MODA application is written in C++. All the experiments are run on a machine with dual 3.4GHz Intel Xeon CPUs and 4.0 GB of RAM running Fedora Core 6. The memory settings for PostgreSQL are 100MB for shared buffers, 100MB for temporarily buffers, and 50MB for working memory. Different experiments use different databases and workloads, described in their respective section.

6.2 Evaluating Risk

In our first experiment we evaluate the impact of the risk metric. We use a TPC-H database with scale factor 1 (1GB), but we skew the data generator for TPC-H to introduce correlation among columns (lineitem.quantity, lineitem.discount) and (part.container, part.size) and (partsupp.availqty, partsupp.supplycost).

For our first experiment, we use the 11 queries listed in Appendix B. In Figure 6.1 we show the effects of varying the weight on the risk metric in the MODA design advisor on the 11 synthetic queries for a disk space budget of 2.4GB. The weight on generality is set to 0. Due to correlation in the data, the design chosen when the weight on the risk is 0 is not good. However, by increasing the weight on the risk metric, the design advisor will start shifting toward configurations that are less risky, (i.e. more robust plans), so execution time decreases up to a certain point. After this point the design advisor starts to choose bad designs because it de-emphasizes benefit too much. The point at which the weight on risk is 0 resembles the performance of a traditional design advisor. It is clear that the risk-aware design advisor performs much better than such a traditional design advisor. In Figure 6.2, we show the effects of the risk on the standard 22 TPC-H benchmark queries, given that the design advisor space budget is 2.4GB. We observe that while increasing the weight on risk the execution time decreases until a certain point where it stays fixed. As before, we observe that the risk-aware design advisor performs better than the normal design advisor on the skewed TPC-H database.

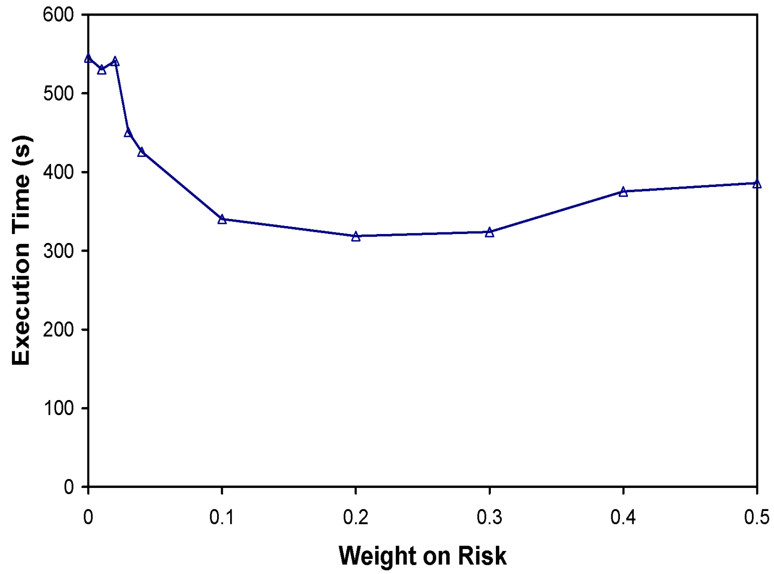

Figure 6.1: Effects of risk on synthetic workload on the TPC-H database

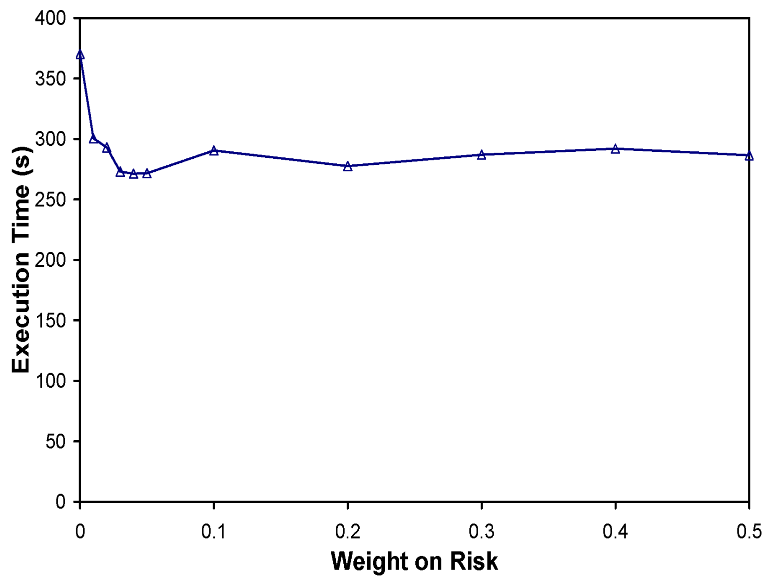

Figure 6.2: Effect of risk on TPC-H benchmark queries

6.3 Generality

To evaluate the generality metric, we use the synthetic workloads left, right, and uniform and the synthetic relation DB_{11C} described in Section 4.3.2.

In Figures 6.3-6.8, we vary the size of the index configuration from two times the database size (1.64 GB) up to five time the database size (4.1 GB). In each experiment we vary the number of statements given to the design advisor from 2 to 100, choosing statements from the left workload. The figures show the number of unique prefixes in the index configuration recommended by the MODA design advisor with the weight of 0 on generality and risk. We observe that as the number of statements given to the design advisor increases, the number of unique prefixes in the recommended configuration increases, which indicates that as the design advisor sees more queries it generates more unique prefixes and becomes less overtrained. The generality metric aims to reduce the overtraining by maximizing the number of unique prefixes even with a small number of queries seen.

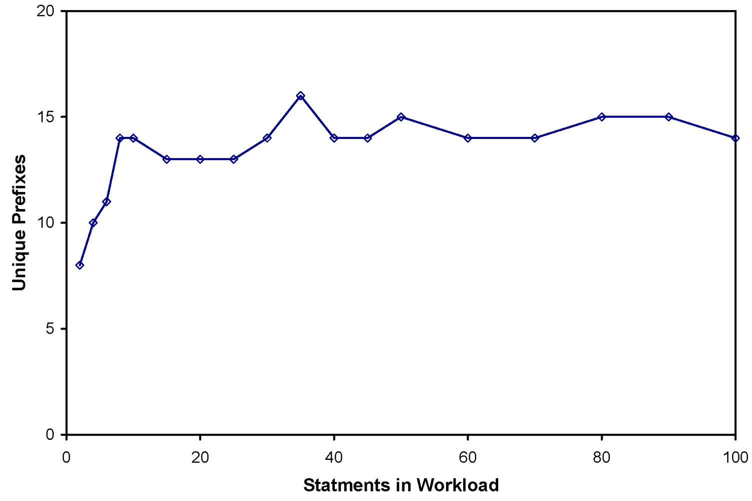

Figure 6.3: Number of unique prefixes in configuration of size $2DB$

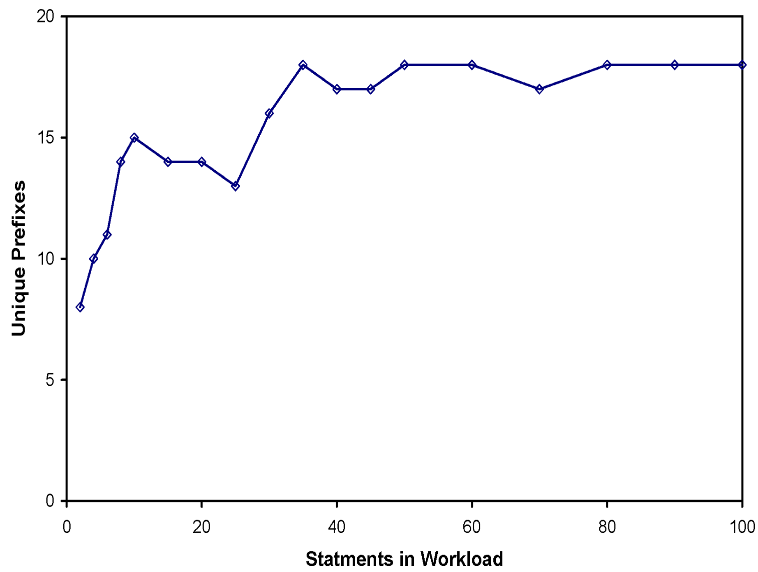

Figure 6.4: Number of unique prefixes in configuration of size $2.5DB$

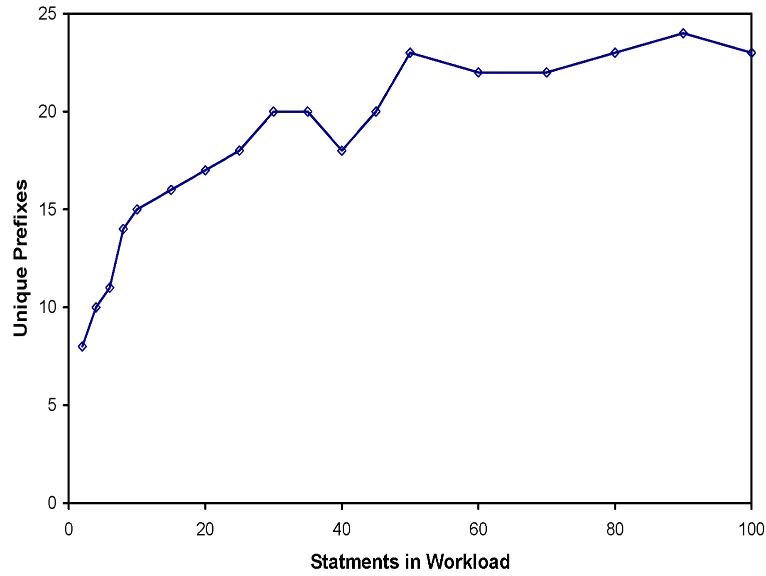

Figure 6.5: Number of unique prefixes in configuration of size $3.5DB$

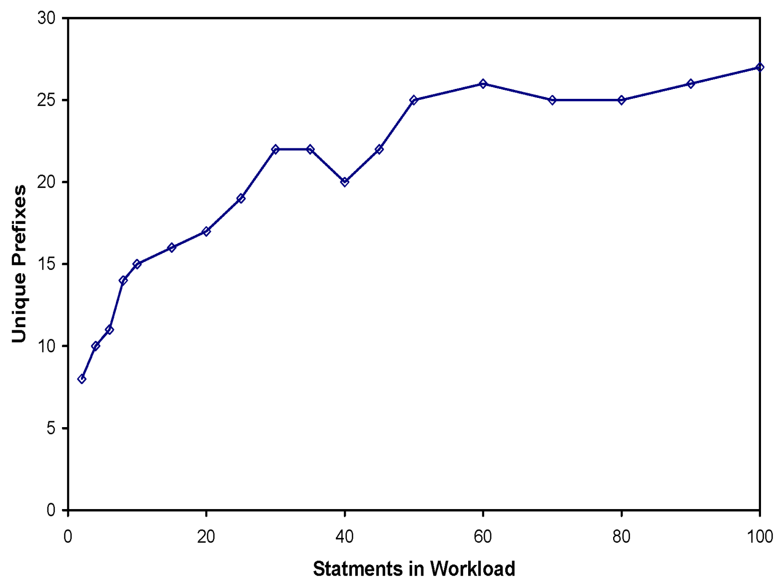

Figure 6.6: Number of unique prefixes in configuration of size $4DB$

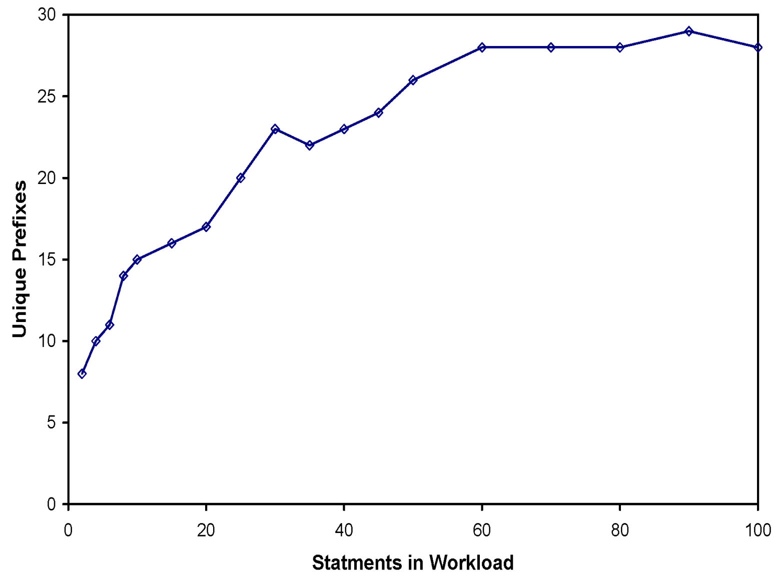

Figure 6.7: Number of unique prefixes in configuration of size $4.5DB$

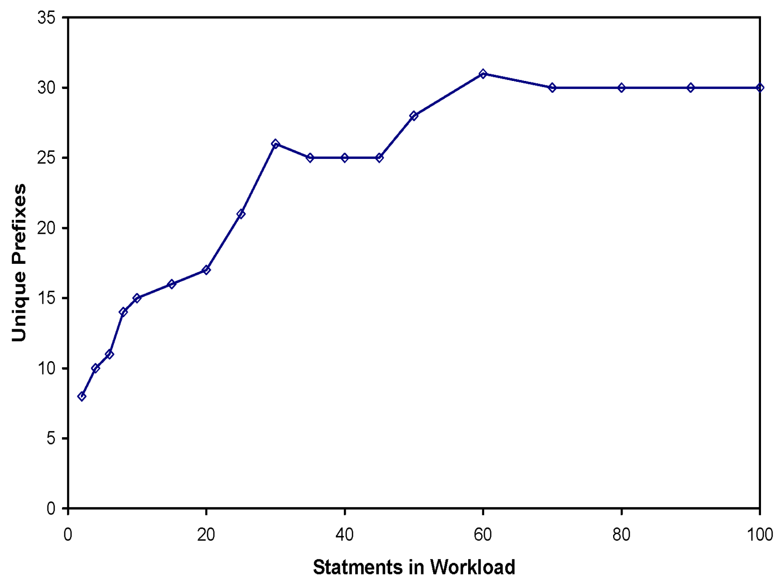

Figure 6.8: Number of unique prefixes in configuration of size $5DB$

In our next experiment, we study the effectiveness of our generality metric on capturing the number of unique prefixes and helping the design advisor maximize it. We vary the index configuration size from 0.5 times to 10 time the database size. We use 100 statements from the left workload. As the size of the configuration increases the number of unique index prefixes chosen by the design advisor increases as shown in Figure 6.9. The figure shows values for five different weights on generality (GW). The weight on risk is set to 0 for all experiments. When the weight on generality is set to 0, the design advisor behaves like a traditional design advisor. As the weight on generality increases the design advisor starts to maximize the number of unique index prefixes more rapidly than the traditional design advisor. In Figure 6.10, we show the value of our generality metric against the size of the configuration. We see that generality grows more rapidly with increasing the weight on the generality metric, which shows that the design advisor is indeed able to maximize this metric and choose more general designs when the weight on generality is increased. Finally, Figure 6.11 shows that there is an almost negligible penalty in the estimated cost compared to the traditional design advisor. Thus, the generality aware design advisor generates more general designs with a minimal sacrifice in performance.

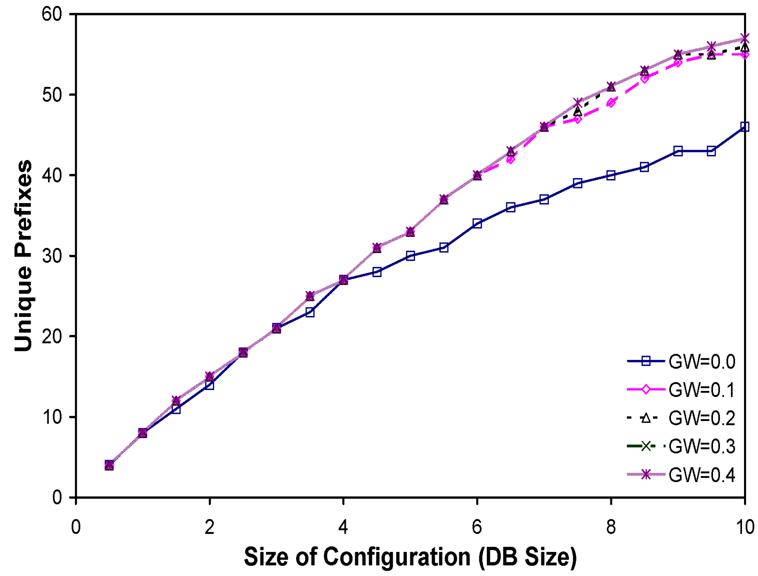

Figure 6.9: Effect of increasing size constraint on number of unique prefixes

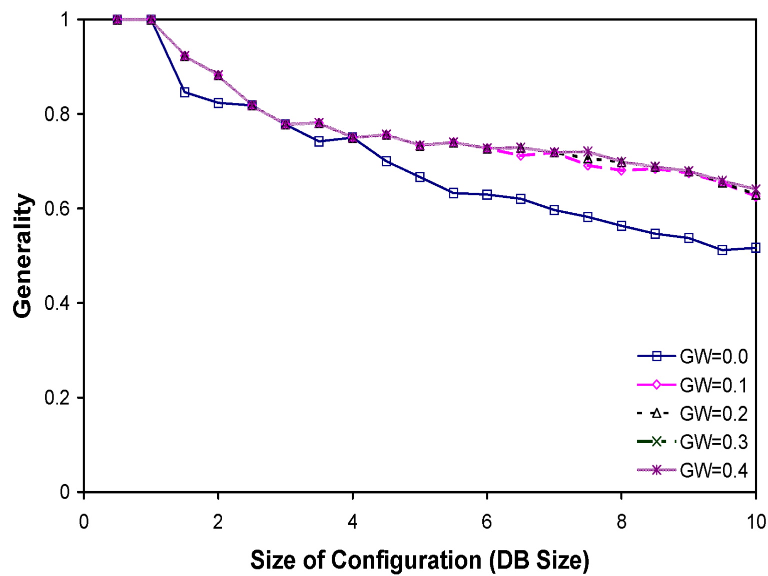

Figure 6.10: Effect of increasing size constraint on generality metric

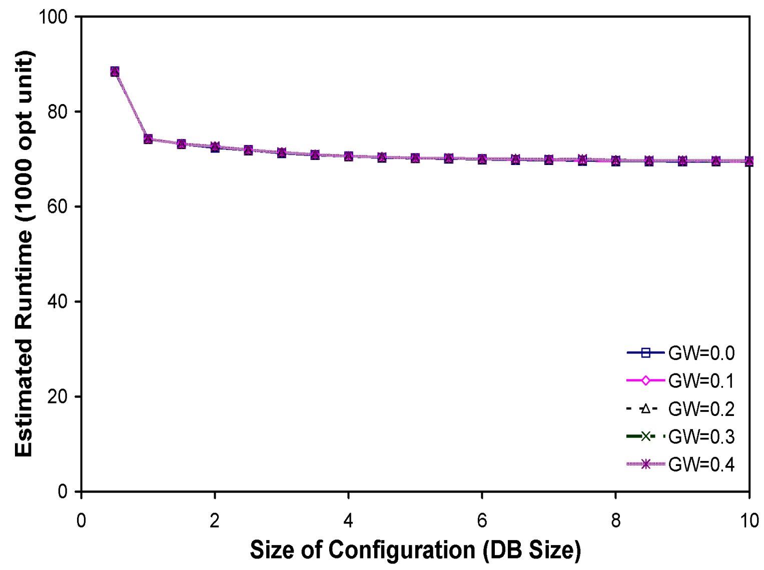

Figure 6.11: Effect of increasing size constraint on performance

In our final experiment, we demonstrate the usefulness of maximizing generality in physical designs. Generality should help the design advisor avoid overtraining for the given training workload. To verify this, we train the design advisor with a left workload consisting of 100 queries, and we test performance with right and uniform workloads of 100 queries. In Figures 6.12 to 6.20 we vary the disk space available for the configuration from 1 times the database size to 5 times the database size in increments of 0.5 the database size. The figures show the estimated runtime for the training and test workloads, varying the weight on generality while keeping the weight on risk 0. As we can see, increasing the weight on generality does not hurt the performance of the training workload but it improves the performance of the test workloads. This becomes more apparent as the disk space budget increases because at the lower disk budget the configuration size is too small to allow for overtraining. Thus we can see that generality helps the design advisor generate physical designs that benefit not only the training workload, but also previously unseen test workloads.

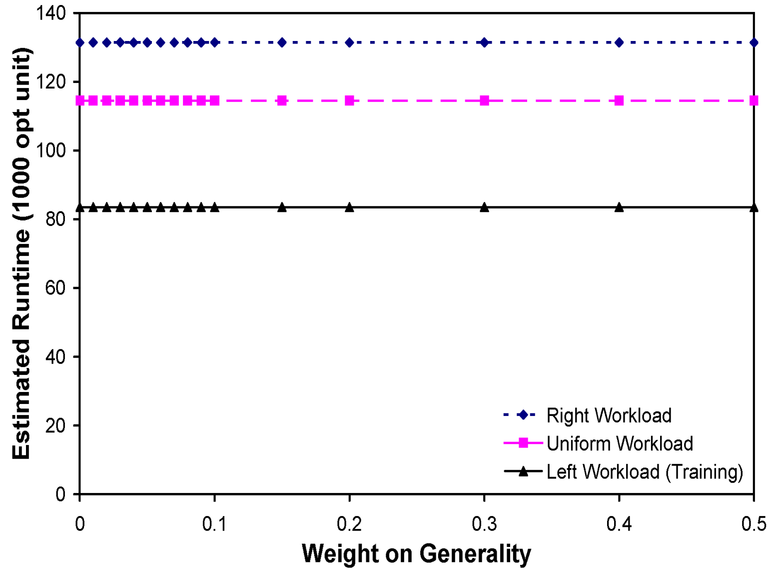

Figure 6.12: Performance on training and test workloads for configuration size 1DB

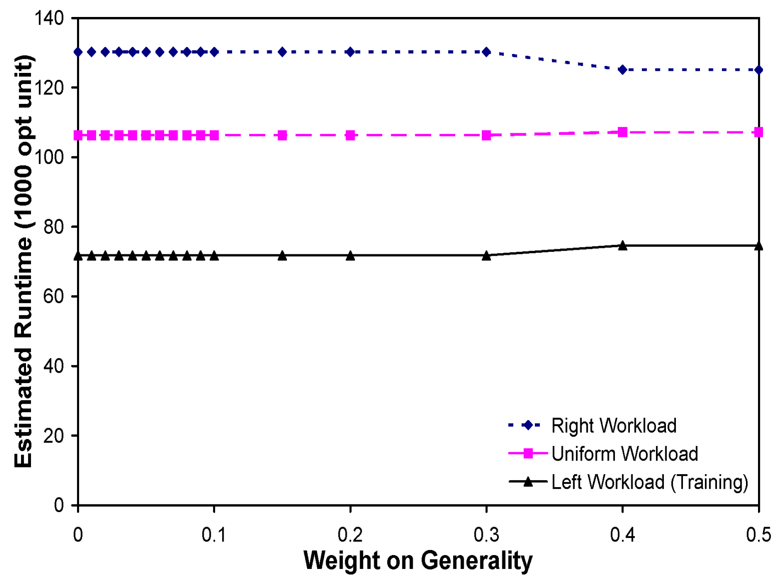

Figure 6.13: Performance on training and test workloads for configuration size 1.5DB

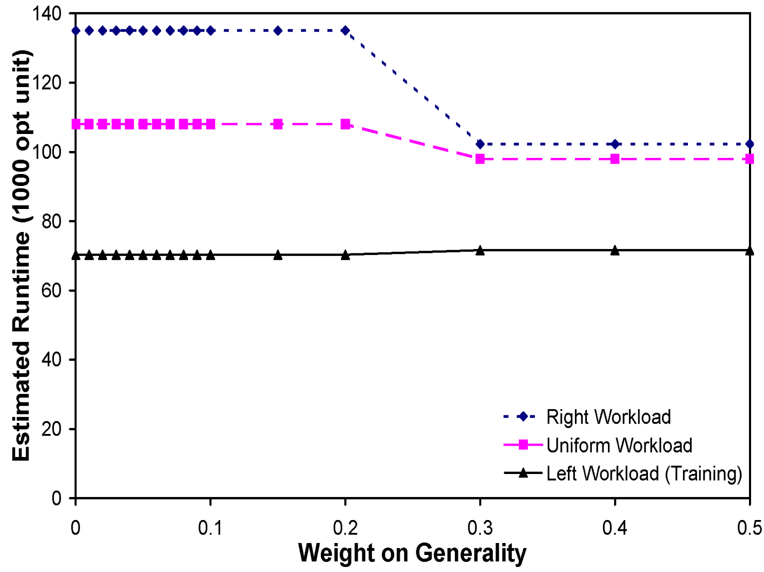

Figure 6.14: Performance on training and test workloads for configuration size 2DB

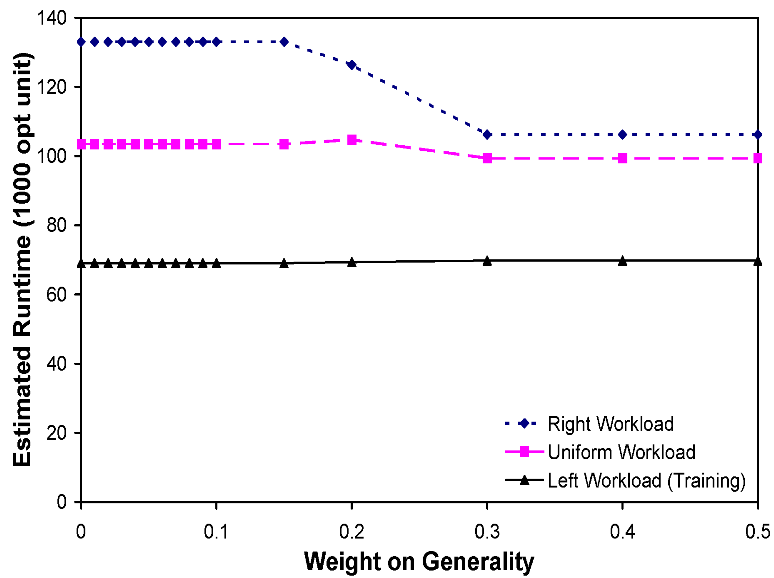

Figure 6.15: Performance on training and test workloads for configuration size $2.5DB$

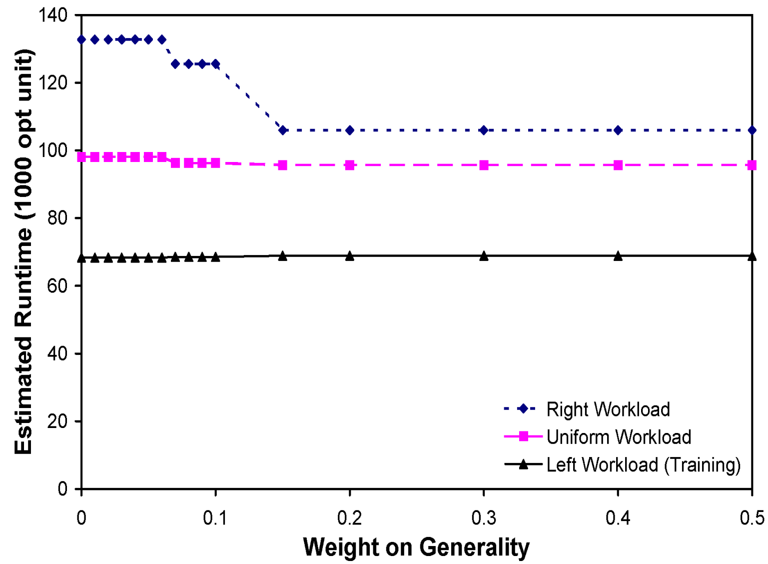

Figure 6.16: Performance on training and test workloads for configuration size $3DB$

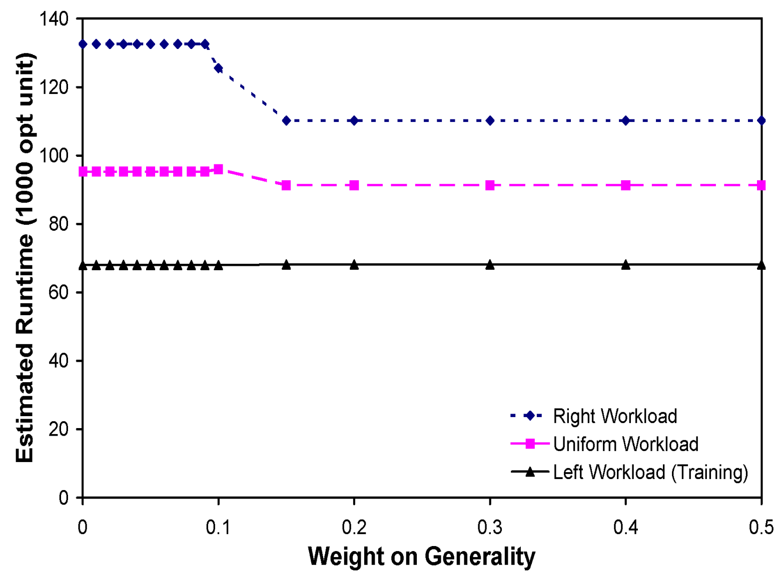

Figure 6.17: Performance on training and test workloads for configuration size $3.5DB$

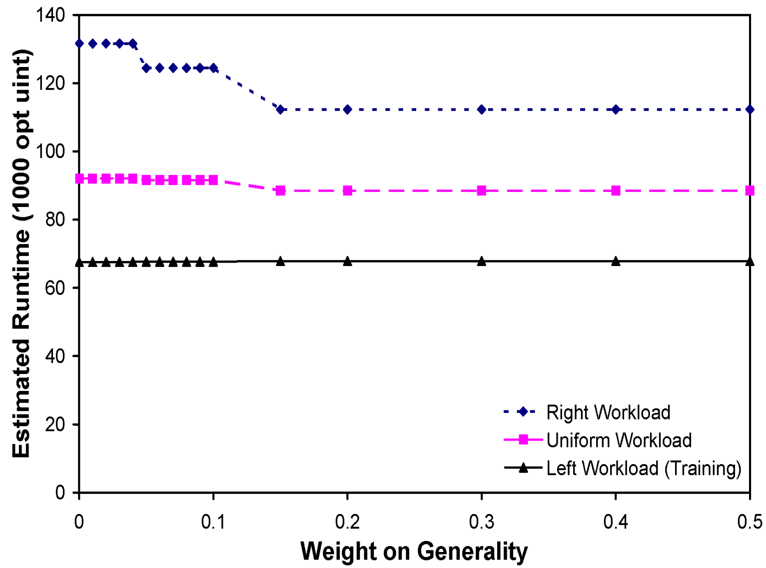

Figure 6.18: Performance on training and test workloads for configuration size 4DB

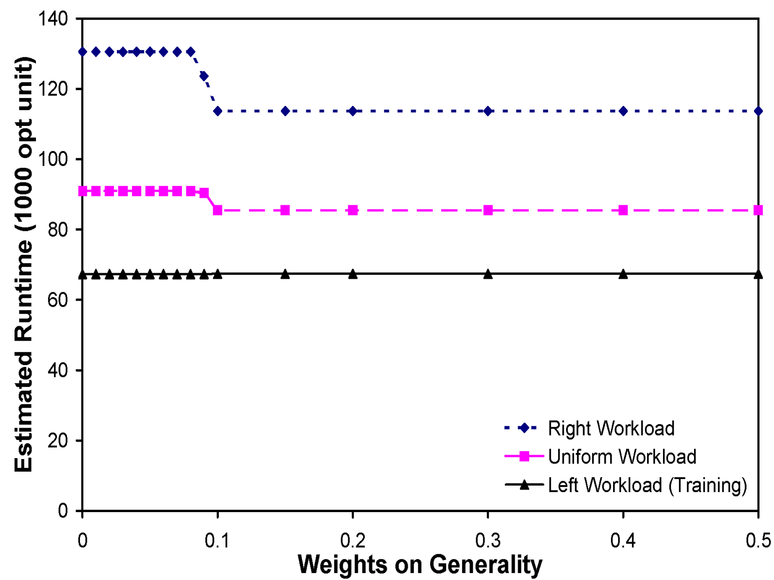

Figure 6.19: Performance on training and test workloads for configuration size $4.5DB$

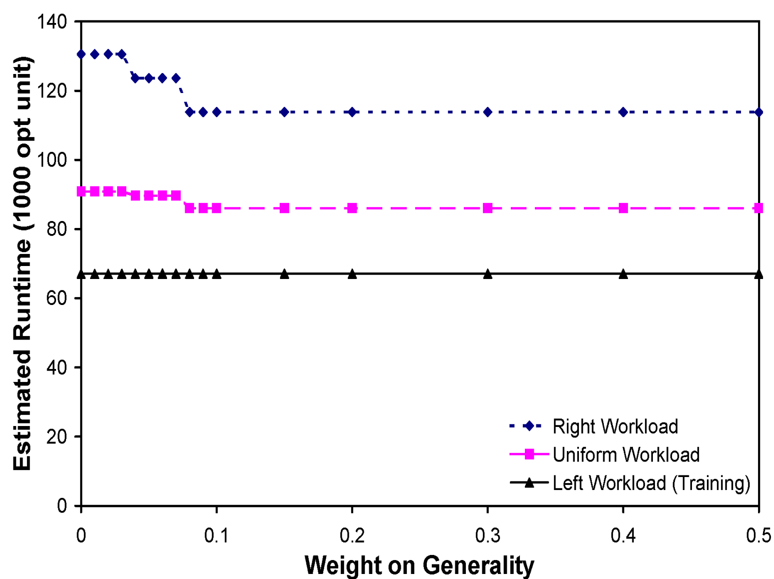

Figure 6.20: Performance on training and test workloads for configuration size $5DB$

Chapter 7

Related Work

7.1 Physical Design Advisor

Physical design advisors employ different techniques for candidate enumeration and search. Some techniques find optimal configurations in polynomial time but under some relaxations to the problem. By removing space constraints and assuming no updates in the workload, [13] guarantees finding an optimal physical design. Work detailed in [28] employs an integer programming technique to find optimal physical designs by only relaxing the space constraint. To find the best configuration without relaxing the problem, the approach adopted in the literature is to prune the number of configurations to approximate the optimal algorithm in reasonable time [20]. The technique detailed in [31] uses the query optimizer to choose all relevant indexes then models the problem as a 0-1-knapsack problem to choose the best set of indexes that satisfy the space constraint. Work shown

in [13] introduces a relaxation based approach for index and materialized views tuning. The tuning advisor starts by selecting the optimal configuration for each query to come up with an optimal overall configuration C^*. Then, to satisfy the constraint on the size of the configuration, the tuning advisor reduces C^* by merging access paths. Index merging is done by fitting indexes with common columns into higher order indexes [21]. MV merging is done by fitting intersecting MVs into other MVs through query rewriting techniques. Another technique starts by merging the physical designs and reducing them into smaller ones that satisfy the space constraint [14].

In [15, 16], this approach is used for online tuning, but the important notion of robustness is not discussed. In [6], the authors introduce a richer model of workloads that captures the temporal aspects of an SQL workload. If the database has more than one workload that are different at different times, the proposed tuning advisor will be able to separately tune the separate workloads instead of finding a design that will benefit the union of all workloads. This diversity in workloads and designs also increases the importance of robustness.

Vertical and horizontal partitioning of relations on disks impacts the performance of queries on databases that exist on multiple disks. Vertical partitioning is projecting different attributes of the relation on different disks. This allows for minimizing the number of pages needed to be read in order to scan the relation. Horizontal partitioning is positioning different portions of the relation on different disks. The partitioning method may be hash based or simply range based. Horizontal partitioning allows for parallelizing disk access

to the same relations. In [4, 7] the authors introduce workload aware techniques that attempt to find an optimal vertical and horizontal partitioning for relations in a database. Studying robustness in the context of physical design aspects beyond indexes such as MVs or partitioning, is an interesting direction for future work.

7.2 Robust Query Optimization

Work shown in [8] formulates the estimated runtime of relational operators in query plans as probability distribution functions based on the selectivity of predicates. The user defines a threshold (a percentage), and the robust query optimizer chooses plans whose costs are guaranteed to be optimal with probability larger than the given threshold.

Indeed a physical design advisor that uses a "what-if" interface to a robust query optimizer would yield robust physical designs. The work shown in [8] provides theoretical guarantees on the query execution cost which may be used to calculate cost guarantees for the entire workload. In our work we choose not to adopt this technique for reducing risk because it is a sampling based technique that will not scale when the optimizer is called a large number of times. The robust query optimizer does not use the database statistics to estimate cardinalities but instead executes the operators on a sample of the database. This may be reasonable when optimizing one query but in a design advisor the query optimizer is called thousands of times and this will not scale. Therefore, we elect to choose a cheaper approach for assessing the potential for errors in a query plan.

7.3 Workload Compression

Many index tuning techniques employ a simple approach for compressing workloads by choosing the most expensive $X\%$ of the queries [5, 20, 24, 33]. This approach has inherent problems. For example, queries need to be costed before determining the physical design, while the cost of a query is highly dependent on the physical design. Moreover, in the case of skewed workloads, where just a few queries are in the $X\%$, many statements will be pruned out. This may cause the design advisor to miss many candidate columns or even tables. More principled approaches are given in [18, 27]. In [18] the authors introduce a generic workload compression technique that may certainly be applied to physical design tuning. In this approach one provides a single distance function between two statements. Statements are then partitioned using the $K - meanoid$ algorithm to yield a reduced set of centers. Similar statements are placed in the same partition, while statements of low similarity are placed in different partitions. Intuitively, picking the center of each partition would give a good representative of the partition. The algorithm achieves higher degrees of accuracy through further optimizations. In [27] a sampling technique is employed to choose a candidate subset of the workload that represents the entire workload. In this work, a statistical primitive for evaluating the confidence that the sample is representative of the whole is used to choose good subsets (i.e., compressed workloads) in a short time. Finally, the representative workload produced has a conservative probabilistic guarantee that the estimated runtime of the configuration tuned on the compressed workload will be no larger than the performance of the configuration tuned on the original workload by

certain threshold defined by the user.

7.4 Automatic Statistics Collection

Data distribution statistics are critical for accurate cost estimation and query optimization. Automatic collection and maintenance of data distribution statistics is an active area of research. Collecting all possible multi-column statistics is infeasible. Therefore, workload aware techniques are employed to analyze queries and identify a reduced set of critical multi-column statistics. Intuitively, the critical multi-column statistics are those that produce estimates that differ most in accuracy for the given workload from estimates produced without the statistics. Many statistics collection techniques (e.g. [10, 11, 12]) use a variant of the *Extreme Cardinality Estimation* technique [22]. In the Extreme Cardinality Estimation technique the workload is evaluated once using no statistics and this forcing the query optimizer to rely on "magic numbers". The sensitivity of the different cardinalities and costs to the values of these magic numbers is analyzed, and the most sensitive cases indicate the statistics that are most critical to have. In addition to the work on determining *which* statistics to collect, there is also work on *when* to collect statistics [3].

All this work in the area of automatic statistics collection aims at reducing errors in the cardinality estimates made by the optimizer. Recall that our risk metric evaluates the potential for choosing bad plans due to erroneous cardinality estimates. Thus, as the cardinality estimates become more accurate, the riskiness of different physical designs is

reduced. In the limit, if the optimizer makes no mistakes, there is no need for measuring or minimizing risk in physical designs. However, we argue that optimizers will always make errors since they rely on a model of the data distributions that is necessarily lossy (since we want to minimize the space required for the model). Thus, while work on automatic statistics collection may reduce the riskiness of physical designs, it will never eliminate it completely.

Chapter 8

Conclusion and Future Work

As we increasingly move towards database systems with truly zero administration, physical design advisors need to make recommendations that are more robust, since they will be implemented without being verified by a DBA. In this thesis, we present two dimensions for measuring the robustness of a physical design and two metrics for evaluating the quality of a physical design along these dimensions.

We introduced the risk metric that assess confidence in the query optimizer costing of the benefit of an index to a query workload. This metric is based on the MAXE cost that estimates the cost of a plan under the worst case cardinality assumptions. The wider the gap between the estimated cost and the MAXE cost, the lower the confidence in query optimizer costing, and hence the higher the risk. We use the risk metric to assess the robustness of index configurations to query optimizers errors.

Our second metric is the generality metric that assess the robustness of an index config-

uration to changes in the query workload. The generality metric penalizes configurations with high degrees of redundancy in index prefixes. We show that traditional index advisors that focus solely on minimizing estimated runtime can overtrain on the given workload. These index advisors may choose indexes with redundant index prefixes as long as these indexes benefit workload queries even slightly. If there is enough diversity in the workload queries, this overtraining will be reduced. Our generality metric aims at reducing this overtraining even for small workloads with low diversity. This makes the recommended physical design useful not only for the training workload, but also for previously unseen queries that come in the future.

To incorporate risk and generality into a design advisor, the main change required is to modify its objective function to include these two quality measures in addition to the traditional quality measure of benefit to workload queries. We present a multi-objective cost function that combines the three metrics using a weighted sum, where the weights indicate the relative importance of the different quality dimensions. We have implemented a simple design advisor that uses our cost function and performs a greedy search (modified to account for index interaction) on an exhaustive space of candidate indexes, and we have implemented the required server-side extensions for our design advisor in PostgreSQL. We show experimentally using this implementation that our approach does indeed generate designs that are more robust.

8.1 Future Work

8.1.1 More Types of Robustness

In this thesis we have examined robustness to specific types of errors and workload changes. It would be interesting to expand our work to include other notions of robustness such as protecting against errors made by the optimizer for reasons other than cardinality estimation inaccuracy, robustness to changes in query frequencies in the workload, and even robustness to fundamental changes in the query templates.

8.1.2 Interaction With Statistics

In this thesis we have assumed that the query optimizer does not have multi-column statistics or join statistics. Many database systems are moving to include workload aware collection of multi-column and join statistics. The MAXE cost could be adapted to work with such systems. If the statistics needed to evaluate the cost of an operator are available in the system, the MAXE costing can be the same as the estimated cost. MAXE cost would be used only to counterbalance assumptions made by the query optimizer when the statistics are not present. Another, interesting direction is to make the choice of which statistics to collect depend on the statistics that would maximize the confidence in the selected physical design.

8.1.3 Robustness in Online Automatic Physical Database Design

Online techniques for physical design tuning such as [16] assume zero database administration, which necessitates the use of robust tools that provision for query optimizer errors and query workload changes. It would be interesting to examine the effects of using techniques similar to what we presented in this thesis to augment online tuning tools.

8.1.4 When to Re-tune?

In [15], the authors raise the problem of how to automatically decide when to re-tune the physical design of the database. The technique relies only on monitoring changes in the estimated runtime benefit. It would be useful to include our benefit and risk metrics when deciding when to re-tune.

8.1.5 Materialized Views and Physical Data Layout

This thesis does not tackle issues concerning choosing materialized views (MVs) and physical data layout but this is very relevant as future work. Risk and generality metrics may be used to assess the quality of a set of indexes and materialized views. The risk metric will examine the query optimizer‘s confidence in costing the queries in the workload. The generality metric will prevent design advisor overtraining, e.g., choosing very specific MVs that exactly match the queries in the workload. The problem of generalizing MVs is very relevant when provisioning for future workload changes. Risk and generality may also be

important in the case of deciding physical data layout such as, vertical and horizontal partitioning of data on disks.

Bibliography

[1] PostgreSQL. http://www.postgresql.org.

[2] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1999.

[3] Ashraf Aboulnaga, Peter J. Haas, Sam Lightstone, Guy M. Lohman, Volker Markl, Ivan Popivanov, and Vijayshankar Raman. Automated statistics collection in DB2 UDB. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[4] Rakesh Agrawal, Surajit Chaudhuri, Abhinandan Das, and Vivek R. Narasayya. Automating Layout of Relational Databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003.

[5] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for Microsoft SQL Server

2005. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[6] Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic physical design tuning: Workload as a sequence. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2006.

[7] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[8] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[9] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[10] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[11] Nicolas Bruno and Surajit Chaudhuri. Efficient creation of statistics over query expres-

sions. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2003.

[12] Nicolas Bruno and Surajit Chaudhuri. Conditional selectivity for statistics on query expressions. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[13] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[14] Nicolas Bruno and Surajit Chaudhuri. Physical design refinement: The "merge-reduce" approach. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2006.

[15] Nicolas Bruno and Surajit Chaudhuri. To tune or not to tune? A lightweight physical design alerter. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2006.

[16] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[17] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2001.

[18] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. Compressing SQL workloads. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[19] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1998.

[20] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1997.

[21] Surajit Chaudhuri and Vivek R. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.

[22] Surajit Chaudhuri and Vivek R. Narasayya. Automating statistics management for query optimizers. *IEEE Trans. Knowl. Data Eng.*, 13(1):7–20, 2001.

[23] Surajit Chaudhuri and Gerhard Weikum. Foundations of automated database tuning. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.

[24] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL tuning in Oracle 10g. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

[25] Amr El-Helw, Ihab Ilyas, Wing Lau, Volker Markl, and Calisto Zuzarte. Collecting and maintaining just-in-time statistics. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

[26] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, 1988.

[27] Arnd Christian König and Shubha U. Nabar. Scalable exploration of physical database design. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.

[28] Stratos Papadomanolakis and Anastassia Ailamaki. An integer linear programing approach to database design. In *Proceedings of the International Workshop on Self-Managing Database Systems (SMDB)*, 2006.

[29] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[30] Transaction Processing Performance Council. "TPC-H, the Ad-hoc Decision Support Benchmark". http://www.tpc.org.

[31] 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 *Proceedings of the International Conference on Data Engineering (ICDE)*, 2000.

[32] Daniel Zilio, Sam Lightstone, Kelly Lyons, and Guy Lohman. Self-managing technology in IBM DB2 universal database. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2001.

[33] 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 *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004.

Appendix A

Changes in PostgreSQL Optimizer

A.1 Modified Data Structures in postgres.h

1. typedef struct

   {

   double sel;

   double worst;

   }Selectivity;

2. typedef struct

   {

   double rows;

   double wroes;

}Rows;

3. typedef struct

   {

   int64 cost;

   int64 wcost;

   }Cost;

4. typedef struct

   {

   int64 rows;

   int64 wroes;

   }Count;

5. typedef struct

   {

   long groups;

   long wgrops;

   }Groups;

A.2 List of Affected Header files

This is the list of the header files changed in the PostgreSQL source code.

1. **include/**
   (a) postgres.h
2. **include/optimizer/**
   (a) cost.h
   (b) plancat.h
   (c) planmain.h
3. **include/utils/**
   (a) selfuncs.h
4. **include/executor/**
   (a) nodeHash.h
5. **include/nodes/**
   (a) plannodes.h
   (b) relation.h
   (c) nodes.h
6. **src/bin/psql/**
   (a) sql_help.h

A.3 List of Files Changed in PostgreSQL

This is the list of files changed in the PostgreSQL source code.

1. **backend/optimizer/geqo/**
    (a) geqo_pool.c
2. **backend/optimizer/util/**
    (a) restrictinfo.c
    (b) pathnode.c
    (c) clauses.c
    (d) plancat.c
3. **backend/optimizer/util/adt/**
    (a) selfuncs.c
4. **backend/optimizer/path/**
    (a) allpaths.c
    (b) clausesel.c
    (c) costsize.c
    (d) indxpath.c
    (e) orindxpath.c

5. **backend/optimizer/plan/**

    (a) createplan.c

    (b) planmain.c

    (c) planner.c

    (d) planagg.c

    (e) subselect.c

6. **backend/commands/**

    (a) explain.c

7. **backend/executor/**

    (a) nodeHash.c

    (b) nodeHashjoin.c

    (c) nodeAgg.c

    (d) nodeSubplan.c

8. **backend/executor/**

    (a) nodeAgg.c

9. **backend/nodes/**

    (a) print.c

   (b) outfuncs.c

10. **backend/catalog/**

   (a) index.c

11. **backend/parser/**

   (a) parser.c

12. **backend/tcop/**

   (a) postgres.c

A.4 Calculating the MAXE Selectivity

In Section A.1, we list the added data structures that we have used to calculate the MAXE cost. The **Selectivity** data structure is used by the PostgreSQL optimizer to store selectivity of predicates. Selectivity estimation is done mainly in the routines found in the files named **clausesel.c** and **selfuncs.c**. We have modified the routines in file **clausesel.c** to generate the traditional selectivity as well as the MAXE selectivities. We have created the selectivity data structure as a C++ **struct** type containing two doubles, one to store the traditional selectivity estimation and one to store the MAXE selectivity estimation. Figures A.1 and A.2 show a code snippet from the file **clausesel.c** where the combined selectivity of predicate list are calculated. Figures A.3 and A.4 show the modified code we use to calculate the MAXE selectivity estimation.

```
foreach (l,clauses)
{
    .
    .
    s1 = s1 * s2;
    .
    .
}
```

Figure A.1: Original code to calculate the selectivity of a predicate list

```
foreach (l,clauses)
{
    .
    .
    selec = selec * subselec;
    .
    .
}
```

Figure A.2: Original code to calculate the selectivity of a predicate list on an index

```
foreach (l,clauses)
{
    .
    .
    s1.sel = s1.sel * s2.sel;
    //Newly added line
    s1.worst = MIN (s1.worst,s2.worst);
    .
    .
}
```

Figure A.3: Modified code to calculate the MAXE selectivity of a predicate list

```
foreach (l,clauses)
{
     .
     .
     selec.sel = selec.sel * subselec.sel;
     //Newly added line
     selec.worst= MIN (selec.sel,subselec.sel);
     .
     .
}
```

Figure A.4: Modified code to calculate the MAXE selectivity of a predicate list on an index

A.5 Calculating the MAXE cost

The selectivity of predicates is used by other routines to calculate the numbers of rows satisfying these predicates. Therefore we have created the **Rows** data structure that contains two double types, again one double type for each kind of costing. We store the number of rows calculated based on the traditional selectivity estimation and the number of rows calculated based on the MAXE selectivity estimation in this structure. The number of rows are mainly calculated at the file named **costsize.c**. The number of rows satisfying a predicate are then used to estimate the runtime cost of an operators. Therefore, we have created **Cost** data structure with two double types, one to store the cost based on the traditional number of rows estimate and the other based on the MAXE estimate. In Figure A.5, we show a code snippet of the modified code in file **costsize.c** used to calculate the number of rows and cost based on the MAXE selectivity.

In order not to affect the traditional query planing that happens mainly in routines

```
tuples_fetched.rows = clamp_row_est(indexSelectivity.sel * baserel->tuples.rows);
    //Newly added line
tuples_fetched.wroes = clamp_row_est(indexSelectivity.worst * baserel->tuples.wroes);
.
.
if (nbytes.cst > work_mem_bytes)
{
startup_cost.cost += seq_page_cost * npages.cost;
run_cost.cost += seq_page_cost * npages.cost;
}
    //Newly added if block
if (nbytes.wcost > work_mem_bytes)
{
startup_cost.wcost += seq_page_cost * npages.wocst;
run_cost.wcost += seq_page_cost * npages.wcost;
}
```

Figure A.5: Modified code to propagate the MAXE cost

found in the file **planmain.c**, all the decisions made by the optimizer are based on the traditional cost. In other words, the query optimizer is oblivious of the MAXE costing. Finally, the structures **Count** and **Groups** are used to estimate the number of unique rows and the of groups satisfying a predicate when optimizing aggregation queries. Optimization of these aggregate queries happens mainly in the file names **planagg.c**.

Appendix B

TPC-H Variant Workload

```
select sum (l_extendedprice)

from lineitem

where l_shipdate > date '1998-01-01';


select sum (l_extendedprice)

from lineitem

where l_commitdate > date '1998-01-01';


select sum (l_extendedprice)

from lineitem

where l_receiptdate > date '1998-01-01' ;
```

```
select sum (l_extendedprice)
from lineitem
where l_shipdate > date '1998-09-01' and l_commitdate >  '1998-09-01';

select sum (l_extendedprice)
from lineitem
where l_shipdate > date '1998-10-17' and l_commitdate >  '1998-10-17';

select sum (l_extendedprice)
from lineitem
where l_receiptdate > date '1998-09-01' and l_commitdate > date '1998-09-01';

select sum (l_extendedprice)
from lineitem
where l_receiptdate > date '1998-10-22' and l_commitdate > date '1998-10-22';


select sum (l_extendedprice)
from lineitem
where l_shipdate > date '1998-09-01' and l_receiptdate > date '1998-09-01' ;
```

```
select sum (l_extendedprice)
from lineitem
where l_shipdate > date '1998-11-14' and l_receiptdate > date '1998-11-14';

select sum (l_extendedprice)
from lineitem
where l_receiptdate > date '1998-09-01' and l_commitdate > date '1998-09-01'
and l_shipdate > '1998-09-01';

select sum (l_extendedprice)
from lineitem
where l_receiptdate > date '1998-07-07' and l_commitdate > date '1998-07-07'
and l_shipdate > '1998-07-07';
```