Robustness in Automatic Physical Database Design

Kareem El Gebaly*
Ericsson
kareem.gebaly@ericsson.com

Ashraf Aboulnaga
University of Waterloo
ashraf@cs.uwaterloo.ca

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. In this paper, we argue that minimizing estimated execution time alone can lead to designs with inherent problems. In particular, if the optimizer makes an error in estimating the execution time of some workload queries, then the recommended physical design may actually harm the workload instead of benefiting it. 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 metrics to evaluate the quality of a proposed physical database design, and we show one way of extending the objective function being optimized by a generic physical design advisor 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.

1. INTRODUCTION

The field of automatic physical database design has received a lot of attention in recent years (e.g., [5, 13, 19, 25]), and most commercial database systems now include tools (or "design advisors") that help the database administrator (DBA) with the task of choosing the best physical design for a given database and query workload [4, 20, 26]. These advisors recommend different physical design features such as indexes, materialized views, data layout, and partitioning, thereby significantly reducing the effort required for database tuning.

*This work was done while the author was at the University of Waterloo.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

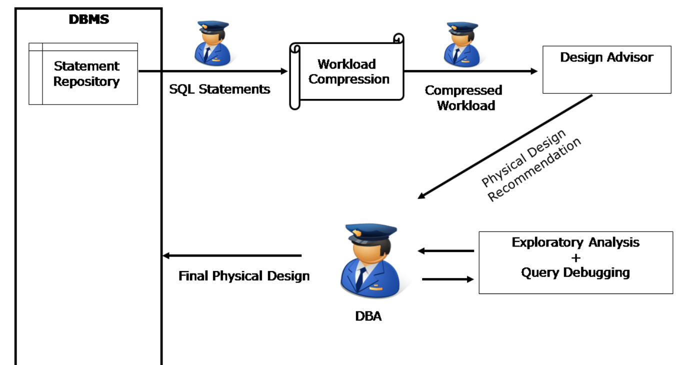

Figure 1: Typical tuning scenario.

Unfortunately, these design advisors can recommend physical designs that are not *robust*. If the assumptions on which the advisor bases its decisions are not valid, or if the workload used for training is not representative, then the chosen physical design may turn out to hurt performance rather than helping it. This has not been viewed as a major problem since the DBA is typically involved in all phases of the physical design process, from workload compression [17], to physical design recommendation, to analysis and debugging [22] (Figure 1). Physical design tools help with all these phases, but the final decision belongs to the DBA.

Recently, there is a trend towards on-line physical design tuning [15], in which the DBA is more and more pushed out of the loop and the physical design recommendations make their way directly to the production environment. It is imperative in such a scenario that the automatic physical design recommendations become more robust and less workload sensitive, since there is no opportunity for the DBA to vet these recommendations.

In this paper, we argue that the current formulation of the automatic physical design problem often leads to recommendations that are not robust. Even using exhaustive search algorithms, we may get physical design recommendations that are not robust because of two main reasons: (1) the query optimizer cost estimates used for computing

the benefit of a physical design may be inaccurate, and (2) the production workload may be different from the training workload. We slightly modify the formulation of the automatic physical design problem to incorporate robustness as a goal, and we define metrics for measuring robustness that can be used effectively in our problem formulation. We focus on making *index recommendation* more robust. Adding robustness to other aspects of automatic physical design, such as recommending materialized views or partitioning, is a subject of future work.

The first kind of robustness we consider in this paper is *robustness to query optimizer cost estimation errors*. All design advisors that we are aware of rely on query optimizer cost estimation to evaluate the benefit of candidate physical designs, augmenting the optimizer with "what-if" interfaces to create hypothetical physical designs and basing physical design decisions on query optimizer cost estimates [18]. Query optimizer costing is far from being accurate, especially when attributes have correlated data distributions. Mitigating the effect of cost estimation errors due to correlation on query optimization is an active field of research [7, 8, 23]. However, optimizers are likely to continue to make cost estimation errors in the presence of correlation, and there is no physical design advisor that is robust against these errors. In this paper we characterize the effect of query optimizer costing errors in terms of *physical design risk*. We define a *Risk* metric that quantifies the robustness of index configurations to optimizer errors, and we integrate this metric into the design advisor.

The second kind of robustness we consider in this paper is *robustness to changes in the workload*. Design advisors assume that the workload used for choosing the physical design (which we call the *training workload*) is representative of the workload in the production environment (which we call the *production workload*). Hence, the goal of the advisor is to find the physical design that minimizes the estimated execution cost of the training workload. If the production workload is even slightly different from the training workload, it may not benefit from the chosen physical design. In this sense, the physical design is *overtrained* to the training workload and lacks *generality*. We define a *Generality* metric and integrate it into the design advisor to get physical designs that are more robust.

Our contributions in this paper are as follows:

- Modifying the formulation of the automatic physical design problem to include robustness as a goal, in addition to benefit in execution cost.
- Introducing a risk metric that quantifies the robustness of a physical design to query optimizer errors.
- Introducing a generality metric that quantifies the robustness of a physical design to workload changes.
- Implementing our proposed metrics in a multi-objective design advisor (MODA) for PostgreSQL, and an experimental evaluation using this implementation.

The rest of this paper is organized as follows. In Section 2, we present an overview of related work. In Section 3, we present our problem formulation. Sections 4 and 5 introduce our Risk and Generality metrics. Section 6 describes the design advisor we implemented for PostgreSQL [1], and Section 7 presents an experimental evaluation of our work using this advisor. We present conclusions in Section 8.

2. RELATED WORK

Physical design advisors employ different techniques for candidate enumeration and search. In [19], candidates are enumerated externally to the optimizer and a greedy algorithm is used for search. The technique in [25] uses the query optimizer to choose all relevant indexes then searches for the best index configuration by solving a 0-1-knapsack problem. Work shown in [12] introduces a relaxation based approach for index tuning. The tuning advisor starts by selecting the optimal configuration for each query to come up with an optimal overall configuration, then reduces the size of this configuration by merging indexes until the space constraint is satisfied. Our work can be used to add robustness to a design advisor that uses any of these approaches.

In [13, 14, 15] an approach for on-line tuning is introduced, but the important notion of robustness is not discussed. In [6], the authors introduce a richer workload model that captures temporal variations in the workload. If the database has more than one workload that are active 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.

One of our goals in this paper is to improve the robustness of physical design recommendations to query optimizer cost estimation errors. Reducing query optimizer cost estimation errors is itself an active area of research. A large body of work in this area is aimed at improving the collection and maintenance of the data distribution statistics used by the optimizer [2, 9, 10, 11, 23]. That work deals both with *which* statistics to collect and *when* to collect the statistics [3]. In [7], the estimated runtimes of relational operators in query plans are modeled as probability distributions, and the optimizer uses these distributions to choose plans whose costs are guaranteed to be optimal with probability larger than a user defined threshold.

All this work in the area of robust query optimization aims at reducing errors in the cardinality and cost estimates made by the optimizer. As these estimates become more accurate, the risk of different physical designs is reduced. In the limit, if the optimizer makes no mistakes, there is no need for quantifying 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 and the time required for query optimization). Thus, while work on automatic statistics collection and robust query optimization may reduce the riskiness of physical designs, it will never eliminate it completely.

The other form of robustness we address in this paper is robustness to workload changes. In physical design literature, finding a "representative" workload is done through workload compression, either by using the most expensive queries as the training workload [4, 19, 20, 26], or by using more principled workload compression approaches [17, 22]. These approaches to workload compression help reduce the complexity of finding a good physical design, but they do not improve the generality of the physical design. If the workload changes, the recommended physical design may not be useful at all.

3. PROBLEM FORMULATION

To estimate the benefit to the query workload of a proposed physical database design, current design advisors rely on a special "what-if" mode of the query optimizer. In this mode, the design under consideration is simulated by inserting the corresponding metadata and statistics into the catalog. The workload queries are optimized with this simulated physical design in place, and the estimated cost of the queries is used to measure the quality of the physical design.

The objective of current design advisors is to maximize the benefit to workload queries of the recommended physical design (which is equivalent to minimizing estimated execution cost), subject to a constraint on the amount of storage available for the physical design. This formulation does not include robustness as a goal.

To add robustness as a goal of the design advisor, we need to change the objective function that is maximized. In this paper, we propose using a simple, weighted multi-objective function, $\rho(.)$, that combines three different metrics to measure the quality of a physical design: a benefit metric, $Benefit(.)$, and two robustness metrics for the two types of robustness that we consider in this paper, $Risk(.)$ and $Generality(.)$. The function $\rho(.)$ combines these metrics in a weighted sum, and is defined as follows:

$$\begin{aligned} &\rho(W, C_O, C_N, q) = \\ &\quad q_1 * Benefit(W, C_O, C_N) + q_2 * Risk(W, C_O, C_N) \\ &\quad + q_3 * Generality(W, C_O, C_N) \end{aligned} \tag{1}$$

The arguments of ρ are a SQL query workload, W, which is the training workload, and 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 configuration to be evaluated. C_O is used as a reference configuration when computing the different quality metrics, and it can be the empty configuration. The q_i's are user specified weights that capture the relative importance of the different metrics to the user. We require that $\sum_{i=1}^{3} q_i = 1$.

Note that adding robustness as a goal does not mean that we ignore the benefit in execution time of candidate physical designs. The execution time benefit is still an important metric in the objective function, but it is now combined with other metrics. Furthermore, adding robustness as a goal does not require us to fundamentally change other components of the design advisor such as candidate enumeration or search. By simply changing the objective function, we obtain physical designs that improve performance and at the same time are robust. We call a design advisor that uses our multi-objective quality function a *multi-objective design advisor*. The goal of such an advisor is to maximize ρ for a given database and workload, subject to a constraint on the available storage. Our definition of ρ has worked well for us in this work. As part of future work, it would be interesting to investigate whether alternative ways of combining benefit and robustness can produce better results.

To be able to easily combine the three quality metrics in the objective function, their ranges must be comparable. Thus, we require the $Benefit$ and $Generality$ metrics to produce values in the range $[0, 1]$, with 0 being the minimum benefit or generality and 1 being the maximum. The $Risk$ metric quantifies a property that we want to *minimize*, so we require it to produce values in the range $[-1, 0]$, with 0 being the least risky and -1 being the most risky. To fit this formulation, we define $Benefit$ as follows:

$$Benefit(W, C_O, C_N) = \frac{Cost(W, C_O) - Cost(W, C_N)}{Cost(W, C_O)} \tag{2}$$

where $Cost(W, C)$ is the cost estimate produced by the query optimizer in "what-if" mode for the total execution time of the workload, W, under physical design configuration C. This formula is based on the assumption that, for a "good" C_N, $Cost(W, C_N) \leq Cost(W, C_O)$. Defining $Risk$ and $Generality$ is a key contribution of this paper, and is the focus of the next two sections.

4. RISK IN PHYSICAL DESIGN

The $Risk$ 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 during cost estimation. 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.

4.1 Motivating Risk

Query optimizers make costing errors for many reasons, but the major cause of error is inaccurate cardinality estimation due to the lack of multi-column statistics on correlated columns [2, 16]. In the absence of such statistics, query optimizers assume that columns are independently distributed, which typically leads to cardinalities being significantly underestimated and to bad query execution plans being chosen [3]. Hence, we focus on this type of error.

First, we show that if the physical design advisor bases its decisions on inaccurate cost estimates, it may recommend physical designs that can hurt performance instead of helping it. In that sense, the physical designs are risky. As an example, we use queries on the TPC-H Benchmark database with scale factor 1 (1GB) [24]. Throughout the paper we use PostgreSQL as our database system (details in Section 7). Consider the following query template:

Q_1:
SELECT AVG (**l_extendedprice**)
FROM LineItem
WHERE l_shipdate BETWEEN $D1$ **AND** $D2$
AND l_receiptdate BETWEEN $D3$ **AND** $D4$.

The two columns **l_shipdate** and **l_receiptdate** in the **LineItem** table are correlated. This may be known to the DBA through domain knowledge, but it is not known to a design advisor. To examine the effect of this correlation on physical design, we vary the dates $D1$–$D4$ to create *high selectivity* time intervals that are 10 days long and *low selectivity* time intervals that are 1 month long. The date ranges we choose for **l_shipdate** and **l_receiptdate** either correspond exactly, in which case the predicates are *positively correlated*, or the **l_receiptdate** interval is before the **l_shipdate** interval, in which case the predicates are *negatively correlated*.

We study the performance of queries with the different combinations of low (L) vs. high (H) selectivity and negative (N) vs. positive (C) correlation. Figures 2 and 3 show the estimated and actual execution costs, respectively, for the different combinations when no indexes exist and when there is an index on (**l_shipdate**,**l_receiptdate**). When the index exists, the query optimizer chooses it as the access path. When there is no index, the query optimizer must

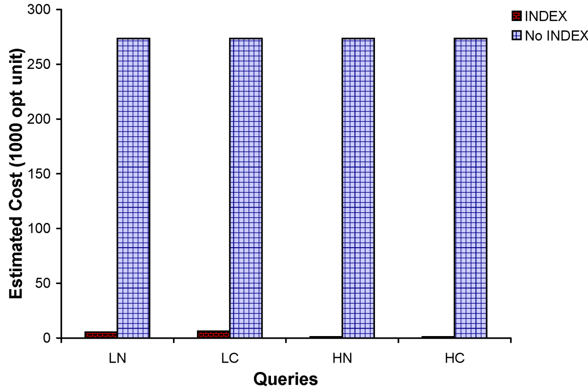

Figure 2: Effect of correlation (estimated cost).

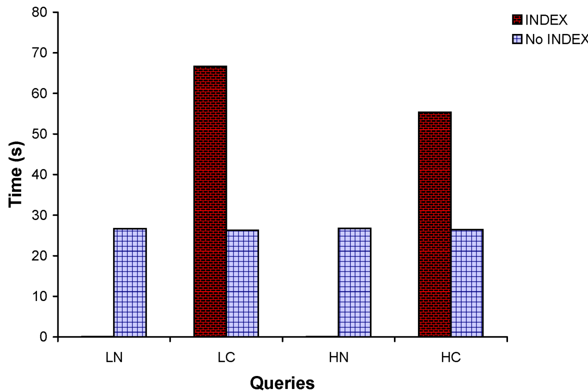

Figure 3: Effect of correlation (actual cost).

choose a sequential scan. From Figure 2, we see that the estimated cost using a sequential scan is significantly higher than the estimated cost using indexes. From Figure 3, we see that the actual execution time of the queries with no correlation is indeed significantly lower using an index versus using sequential scan. On the other hand, the execution time of the correlated queries using an index is *much worse* than the runtime using sequential scan. The query optimizer makes this error because it is assuming that **l_shipdate** and **l_receiptdate** are independent, and this leads to the benefit of the index being estimated 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. Thus, the index is *risky*. The query optimizer and candidate enumeration algorithm of the design advisor 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.

Since the query optimizer tends to overestimate the benefit of risky indexes, the design advisor will choose these 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 may cause a performance degradation instead of improvement. 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 useful disk space budget. Thus, our goal is to avoid such risky indexes. We present our metric for enabling this next.

4.2 Risk Metric

We require the query optimizer to return two cost estimates for each query and not one: 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. Since we focus on the independence assumption, whenever the optimizer needs to estimate the joint selectivity of multi-column predicates we assume full correlation instead of assuming independence. For this pessimistic approach, the joint selectivity of the predicates is the *minimum of the selectivities* and not their product. We call this worst case costing method *Minimal Assumptions eXtreme cost Estimation* or *MAXE cost*.

To support MAXE cost estimation, we need to make changes to the database server to modify the query optimizer so that it returns two costs for each operator: the normal cost and the MAXE cost assuming worst case cardinality for multi-columns predicates. The MAXE cost is always greater than or equal to the normal cost. We have made these changes in the query optimizer of PostgreSQL and we describe them in Section 4.3. The required changes in any other DBMS would be similar.

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 this configuration. This represents an estimate of how much worse the worst case cost could be as compared to the expected cost:

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

Directly using the MAXE gap as a measure of risk for a configuration, C_N, can yield poor results because it ignores the MAXE gap of the original configuration, C_O. Instead, we use the *factor of increase* in the MAXE gap of C_N as compared to C_O: $MAXE\ Gap(W, C_N)/MAXE\ Gap(W, C_O)$. The higher this factor, the more risky the configuration C_N. Since we want the risk metric to range from -1 to 0, with -1 being the most risky, we define our risk metric as follows:

$$Risk(W, C_O, C_N) = \left(\frac{\sum_{q \in W} \left(\frac{MAXE(q, C_O)}{Cost(q, C_O)} / \frac{MAXE(q, C_N)}{Cost(q, C_N)} \right)}{|W|} \right) - 1 \quad (3)$$

If the increase in MAXE gap for configuration C_N is high (i.e., C_N is risky), the ratio inside the summation will decrease, and hence the overall *Risk* metric will move closer to -1. Dividing by $|W|$ is necessary to normalize the summation to the range $[0, 1]$.

4.3 Calculating MAXE Cost in PostgreSQL

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 virtual (hypothetical) index. The PostgreSQL server is allowed to add the index metadata to the catalog but index creation is stopped before actually building the index. The index size is an important statistic, so we make sure it is computed correctly. Virtual indexes can be used by the optimizer like normal indexes, which enables

us to estimate normal plan costs for any configuration.

To estimate MAXE cost, we need to estimate worst case selectivity. This required modifying two routines in which selectivities of multiple predicates are combined:

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.

2. **Cost Bitmap And Node:**
   This routine estimates the overhead of using an index to access a relation. It calculates the cost of retrieving record ids or values from the B-tree index, depending on the selectivity of the predicates using the index.

In both these routines, PostgreSQL assumes that predicates on different columns are independent, so the optimizer multiplies the selectivities of the predicates to evaluate the overall selectivity. This overall selectivity is used to estimate the normal (non-MAXE) 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.

In addition to modifying the way selectivities are estimated, we also needed to change the cost estimation functions of PostgreSQL to return two costs instead of one: the original query optimizer cost, and the MAXE cost. A fundamental change that enables this is changing the selectivity and cost data types in PostgreSQL from **double**s to **struct**s with two members for the two costs. In total, our modifications to PostgreSQL required changes in 35 files and involved 2300 lines of code. More details about the changes are available in [21].

5. GENERALITY IN PHYSICAL DESIGN

The second type of robustness that we consider in this paper is robustness to changes in the workload. State of the art physical design advisors recommend the physical design that minimizes the execution cost of the training workload. This assumes that the production workload will not be different from the training workload. In practice, the training workload is collected from past queries and compiled 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, the query templates and frequencies in the production workload are the same as the training workload but the parameter marker values (the constant values provided to the query) are different. This case is relevant if the queries in the workload come from prepared statements or precompiled applications. In this case, the values bound to the parameter markers 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 templates and parameter markers do not change in a way that affects the desired physical design, but the relative frequencies of the queries in the workload do change. This happens when the mix of application requests in the production workload is different from the mix in the training workload. Having a general configuration with a high number of unique prefixes will also provide robustness to this type of change.

3. **Query Template Change:**
   If query templates are not fixed, meaning that each query could be completely different from all previous queries, then the cost-based workload aware approach is not appropriate for index tuning. There is no room for provisioning for this type of change 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.

5.1 Motivating Generality

In this section, we motivate the need for choosing index configurations with a large number of leading columns. We start by demonstrating how overtraining to the given workload can lead to configurations with a small number of leading columns.

Consider the following database and workload. 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 generated independently, so query optimizer estimates are accurate. Throughout our study of generality, we always rely on optimizer cost estimates to evaluate physical designs, and we make sure that the attributes are uniform and independent so that these optimizer estimates are indeed accurate. This is appropriate for the goal of generality, which is to provision for changes in the workload, not for optimizer errors. Thus, using optimizer estimates to study generality is the simplest and most direct approach.

We have generated a sample relation R with 4M rows (455MB on disk). We consider a workload of four queries, given in Figure 4, that has an estimated execution cost of 506K optimizer units on a configuration with no indexes. Let

Index Column Order	Estimated Cost
∅	180,966
a,b,c	48,078
a,c,b	48,078
b,a,c	44,427
b,c,a	44,427
c,a,b	40,159
c,b,a	40,159

Table 3: Estimated cost for Q_2.

Index Column Order	Estimated Cost
∅	180,953
a,b,c	8,900
a,c,b	8,900
b,a,c	24,767
b,c,a	24,767
c,a,b	98,868
c,b,a	98,868

Table 4: Estimated cost for Q_3.

us assume that we have an unbounded disk space budget, so we can recommend any indexes we want. In this case, the optimal configuration that minimizes the estimated execution cost 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 columns in the query predicates and the column order in the index matches the selectivity of the predicates. The estimated execution cost of this configuration is 311K optimizer units. However, there is a lot of redundancy in the recommended indexes, and if the selectivities of the predicates change due to changes in parameter values, no good indexes will be found. The index configuration has only 4 unique leading prefixes, as shown in Table 1. On the other hand, consider the alternative configuration $\{I(R.a), I(R.b, R.a), I(R.c, R.a, R.b), I(R.d, R.a, R.b, R.c)\}$. This configuration would give a suboptimal estimated execution cost of 370K optimizer units (an increase of 19%), but the number of unique leading prefixes in this configuration is 10, as shown in Table 2. Therefore, it benefits a wider class of queries. We can see that the first index configuration is naive and overtrained. An expert DBA would never allow these indexes to reach production unless she has prior knowledge that the production workload will exactly match the training workload.

Our solution to prevent this kind of overtraining is to measure the number of unique leading prefixes in the index configurations and try to maximize it. This may cause a slight reduction in performance but it allows the design advisor to provision for a much wider class of workloads and not overtrain for the given training workload. To illustrate the importance of having many different leading prefixes (i.e., many different index column orders), consider the following two queries on relation R described above:

Q_2:
SELECT AVG (x)
FROM R
WHERE $a < 0.15$ **AND** $b < 0.125$ **AND** $c < 0.1$

Q_3:
SELECT AVG (x)
FROM R
WHERE $a < 0.01$ **AND** $b < 0.1$ **AND** $c < 0.5$

The estimated execution cost of these two queries for different index configurations each containing one index with different index column orders is shown in Tables 3 and 4. The tables shows that index column order does not matter for Q_2. The difference between the worst and the best column orders is 20%. On the other hand, index column order has a significant effect on the performance of Q_3. Choosing the wrong index column order can degrade performance by an order of magnitude. We would like the design advisor to include as many column orders as possible, potentially choosing column orders that are suboptimal where it does not matter, such as for Q_2, so that the required column orders are available when they do matter for a query such as Q_3, even if this query was not part of the training workload. We rely on the fact that indexes can be useful even if the query predicates do not exactly match index columns. For example, an index $I(a, b, c)$ could be used to reduce runtime of queries on columns $\{a, b, d\}$. In general, an index on a set of columns can benefit a query on a different set of columns if the two sets of columns have a common prefix. Hence our focus on increasing the number of prefixes.

A design advisor will typically recommend indexes with an increasing number of unique prefixes as the number of training queries increases. Thus, the design advisor becomes more general and less overtrained as it sees more queries. By defining generality as a goal for the design advisor, we direct it to recommend index configurations with many unique prefixes even without seeing many queries. This allows the recommended configurations to benefit queries beyond those seen in the training workload.

5.2 Generality Metric

We use a simple metric to assess the generality of an index configuration. We know that the higher the number of unique prefixes the better the generality. At the same time, we want the generality metric to return a number in the range $[0, 1]$. Thus, we define the *Generality* metric as follows:

$$Generality(W, C_O, C_N) = \frac{Number\ of\ unique\ index\ prefixes\ in\ C_N}{Number\ of\ possible\ unique\ prefixes\ for\ C_N} \quad (4)$$

The number of unique index prefixes in C_N is the cardinality of the set containing all the index prefixes in the configuration. The number of possible unique index prefixes for C_N is calculated by adding the number of columns in every index. For example, the configuration $\{I(a, b, c), I(b, d)\}$ has a three column index and a two column index. The three column index $I(a, b, c)$ may be used whenever one of the three indexes $\{I(a), I(a, b), I(a, b, c)\}$ would be useful. Thus, the three column index gives us the opportunity to have three unique prefixes. Similarly, the two column index $I(b, d)$ may be used whenever one of the two indexes $\{I(b), I(b, d)\}$ would be useful, so it gives us the opportunity to have two unique prefixes. Therefore the number of possible unique prefixes for this configuration is five. In this case, the actual number of unique prefixes is also five, so the configuration is as general as possible and has a *Generality* value of 1.

Our generality metric is easy to compute without the need

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: Workload for illustrating overtraining.

Indexes	Unique 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 1: Unique prefixes in the optimal configuration.

for any server side extensions. It does not depend on W, which is expected since it is measuring the potential for benefiting queries *beyond* W. It also does not depend on C_O. With the metrics defined in Equations 2–4, we now have all the components of the objective function defined in Equation 1, and we can use this objective function to add robustness to a physical design advisor.

6. MULTI-OBJECTIVE DESIGN ADVISOR

To evaluate the effectiveness of our approach for providing robustness, we have implemented a design advisor that optimizes the multi-objective cost function defined in Equation 1. We call our design advisor *MODA* for *Multi-Objective Design Advisor*. The weights in Equation 1 enable the user to specify the relative importance of the different quality metrics. If the user sets the weight on the *Risk* and *Generality* metrics to zero, MODA will behave like a normal index tuning advisor that does not have robustness as a goal. 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.

In this paper we want to focus on the effects of the proposed robustness metrics on the behavior of the design advisor. Hence, we want the candidate enumeration, search, and benefit estimation of MODA to be as comprehensive as possible. Typical design advisors use different techniques that reduce tuning time but may also reduce the quality of the chosen solution. In particular, design advisors may choose to ignore index interaction and compute the benefit of an index only once regardless of what other indexes exist. Design advisors may also prune the space of configurations searched using heuristics, and these heuristics may prune away the optimal configuration. In MODA, we take index interaction into account and we have minimal pruning. To take index interaction into account, we re-evaluate the benefit of all candidate indexes whenever we add an index to the recommended configuration. To avoid excessive pruning, we only prune syntactically irrelevant index columns and indexes that do not benefit any query in the workload.

Algorithm 1 presents the outline of the MODA index recommendation process. The algorithm takes as input a workload, W, the amount of available storage, S, the current index configuration, C_O, and the weights for the different quality metrics, q_1–q_3. The algorithm starts 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. The algorithm then repeatedly computes the overall quality of every candidate index and greedily chooses the candidate with the highest quality while satisfying the disk space constraint. Candidate indexes are re-evaluated after adding an index to the recommended configuration so that we can take index interactions into account. If a candidate index has zero benefit for all workload queries (or negative benefit due to updates) we remove it from further consideration. The algorithm stops when it exhausts the available disk space budget.

This algorithm employs a greedy search over an extensive set of candidates while taking index interaction into account, which means that it may be slow but it will typically find an index configuration that is close to optimal. This makes it suitable for the purpose of evaluating our metrics. We have implemented the server side extensions required for MODA in PostgreSQL, as described in Section 4. The client side MODA application connects to this modified server and recommends index configurations for a given workload.

7. EXPERIMENTAL EVALUATION

7.1 Experimental Setup

In our experiments we use a PostgreSQL 8.1 server modified as described in Section 4.3. 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. All our experiments were run cold, with the PostgreSQL buffers and the file system cache cleared between experiments.

Different experiments use different databases and workloads, described in their respective sections. We chose to use different databases and workloads because we want to illustrate different kinds of problems that arise due to lack of

Indexes	Unique 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 2: Unique prefixes in the alternative configuration.

Algorithm 1 Multi-Objective Design Advisor - (MODA)

```
MODA(W, S, C_0, q_1, q_2, q_3)
 1  cands ← All candidate indexes generated from W
 2  C_N ← C_0                          ▷ Recommended configuration.
 3  A ← S                              ▷ Available disk space.
 4  candFound ← TRUE                   ▷ Did we add a candidate in the last iteration?
 5  while A > 0 and candFound
 6       do
 7          ▷ Evaluate candidates.
 8          for i ← 0 to |cands|
 9               do
10                  CreateHypotheticalIndex(cands[i])
11                  C_Temp ← C_N ∪ cands[i]
12                  cands[i].size ← Estimated size of hypothetical index cands[i]
13                  cands[i].benefit ← Benefit(W, C_O, C_Temp)
14                  cands[i].risk ← Risk(W, C_O, C_Temp)
15                  cands[i].generality ← Generality(W, C_O, C_Temp)
16                  cands[i].MODA_quality ← q_1 * cands[i].benefit + q_2 * cands[i].risk + q_3 * cands[i].generality
17                  ▷ Prune candidates that do not benefit any query.
18                  if (cands[i].benefit <= 0)
19                    then
20                         Remove cands[i] from cands
21                  DropHypotheticalIndex(cands[i])
22
23          ▷ Choose next candidate.
24          candFound ← FALSE
25          Sort cands descendingly on MODA_quality
26          for i ← 0 to |cands|
27               do
28                  if cands[i].size < A
29                    then
30                         candFound ← TRUE
31                         C_N ← C_N ∪ cands[i]
32                         A ← A − cands[i].size
33                         CreateHypotheticalIndex(cands[i])
34                         Remove cands[i] from cands
35                         break      ▷ Goto 5.
36  return C_N
```

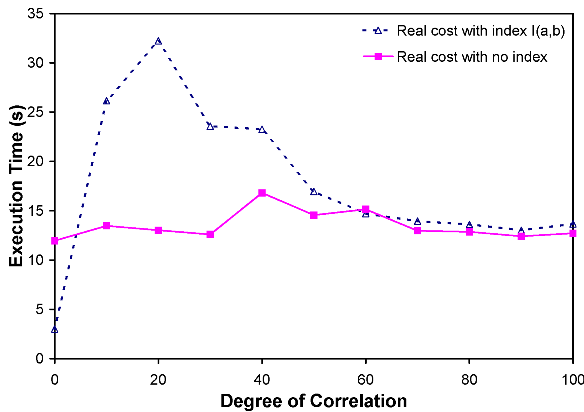

Figure 5: Execution time of Q_4 - Bitmap Index Scan.

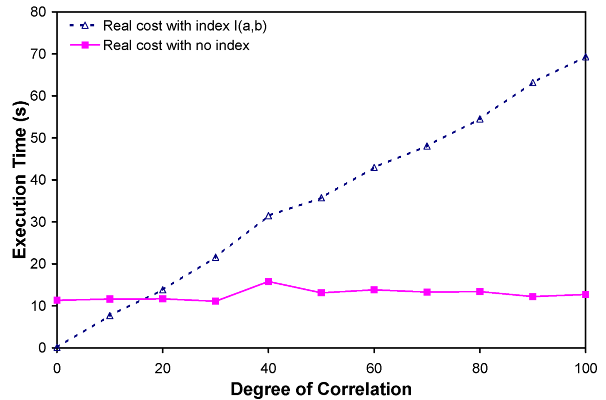

Figure 6: Execution time of Q_4 - Index Scan.

robustness and how well we protect against them. One can view the different problems as different kinds of "bugs" in the physical design. It is possible to devise a single database and workload that illustrates all the different kinds of bugs, but that database and workload will likely be highly contrived and hence will not help us gain any insights. Using different databases and workloads for different experiments helps us gain the required insights.

7.2 Measuring Risk

In our first experiment, we ask how much risk there can be in a physical design, and whether our approach for measuring this risk is effective. Consider the following query template on relation $S(a, b, c)$:

Q_4:
SELECT AVG (c)
FROM S
WHERE a **BETWEEN** $V1$ **AND** $V2$
AND b **BETWEEN** $V3$ **AND** $V4$

We synthetically generate different instances of relation $S(a, b, c)$, each with 16 million rows. To be able to control the degree of correlation we generate the three columns a, b, and c as independent, uniformly distributed random variables in the range $[0 - 1]$. We then choose a percentage of the rows and make their a and b columns equal, thereby introducing correlation. For this experiment, we vary the percentage of rows for which a and b are made equal from 0% to 100%.

Figure 5 shows the actual execution time of Q_4 on relations with varying degrees of correlation when $V1$-$V4$ are chosen to make the selectivity of the predicates on a and b both be 0.1. The query optimizer assumes a and b are independent and so it estimates the combined selectivity of both predicates to be 0.01. Based on this, the optimizer would choose to use an index on (a, b) if one exists, using the Bitmap Index Scan relational operator to access the index. Figure 5 shows the execution time when an index on (a, b) exists and is used by the query and when no index exists so a sequential scan is used. At the level of correlation of 0%, which is the point assumed by the optimizer, using the index is indeed better than using a sequential scan so the decision to use the index is correct. When correlations exist, the sequential scan is always better than the index scan because the index scan performs a lot more I/O so the choice made by the optimizer is wrong. The execution time using the index can be almost 3 times slower than using the sequential scan, due to the riskiness of the index. Note that at higher degrees of correlation the execution time using an index drops and becomes similar to that of a sequential scan because the file system detects the pattern of I/O requests and prefetches the file containing the relation in a sequential fashion.

Figure 6 shows the same experiment as Figure 5, but when $V1$-$V4$ are chosen to make the selectivity of the predicates on a and b both be 0.001. The PostgreSQL query optimizer still chooses to use an index on (a, b) if one exists, but for these more selective predicates it uses an Index Scan operator instead of a Bitmap Index Scan operator. As in the previous experiment, we see that the index is risky, and can degrade performance by a factor of almost 7.

Thus, we can see that even for simple queries, choosing risky designs can lead to significant performance degradation. For the first query, the estimated cost when using the index on (a, b) is $36,034$ optimizer units, while the MAXE cost is $118,839$ optimizer units, leading to a MAXE gap of 3.3. For the second query, the estimated cost using the index is 301 optimizer units while the MAXE cost is $29,410$ optimizer units, giving a MAXE Gap of 97.7. Thus, we can see that our approach to measuring risk does distinguish more risky from less risky designs.

7.3 Effectiveness of the Risk Metric

Next, we turn our attention to whether our *Risk* metric can effectively make recommended physical designs more robust to query optimizer errors. For this experiment, we use our MODA design advisor and two realistic workloads based on the TPC-H Benchmark with scale factor 1 (1GB). For each workload, we run the MODA design advisor on the workload queries giving it a disk space budget that its twice the database size, create the recommended index configuration, and measure the actual execution time of the workload using this configuration. We repeat the process for different values of q_2, the weight on the risk metric in Equation 1. Sine the training workload is the same as the test workload, we set the weight on the generality metric, q_3, to 0. The weight on benefit is set to $q_1 = 1 - q_2$.

For our first experiment in this section, we use a syn-

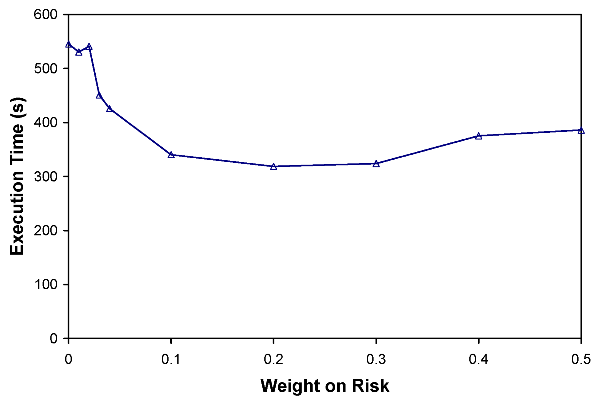

Figure 7: Effectiveness of the risk metric - synthetic queries.

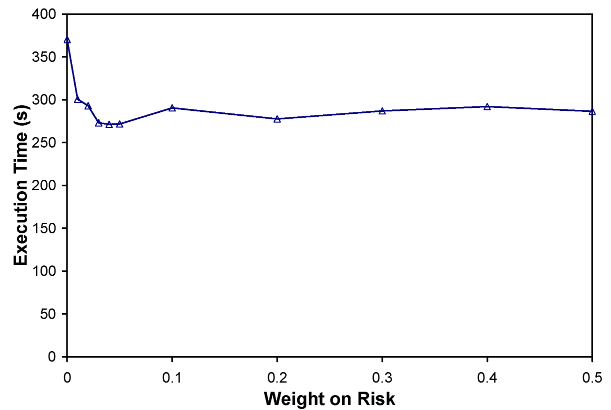

Figure 8: Effectiveness of the risk metric - TPC-H benchmark queries.

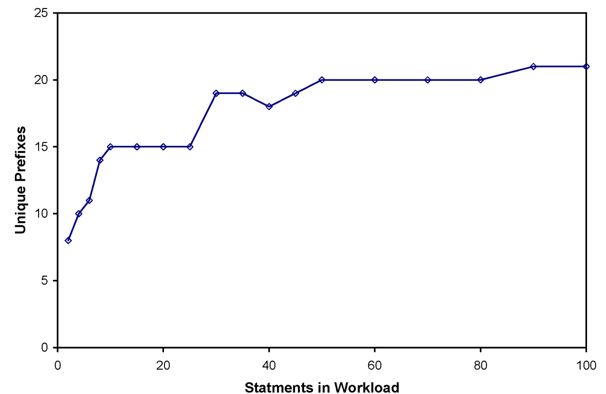

Figure 9: Unique prefixes - $3\times$ DB size

thetic workload on an unmodified TPC-H database. The workload consists of queries with range predicates on the correlated **l_shipdate**, **l_receiptdate**, and **l_commitdate** attributes of the **LineItem** table. A listing of these queries can be found in [21]. Figure 7 shows the execution time of these queries for varying weights on the risk metric in the MODA design advisor. Due to the correlation in the data, the physical design recommended when the weight risk is 0 is sub-optimal. However, by increasing the weight on the risk metric, the design advisor will start shifting towards configurations that are less risky (i.e. more robust) so execution time decreases up to a certain point. After this point the design advisor starts to choose bad designs because it deemphasizes 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 advisor.

For the second experiment in this section, we use the standard 22 TPC-H Benchmark queries, but we modify the TPC-H data generator to introduce correlation among columns (**LineItem.l_quantity**, **LineItem.l_discount**), (**Part.p_container**, **Part.p_size**), and (**PartSupp.ps_availqty**, **PartSupp.ps_supplycost**). Figure 8 shows the execution time of the 22 TPC-H Benchmark queries while varying the weight on risk. As before, we observe that the risk-aware design advisor performs better than the traditional design advisor.

7.4 Evaluating Generality

To evaluate the generality metric, we need a setting that is rich enough so that the test workload can be significantly different from the training workload. To get this, we create a synthetic relation, G, that has 4M rows and 11 decimal columns each generated independently according to a uniform distribution in the range $[0-1]$. This matches the assumptions made by the optimizer and allows us to rely on optimizer estimates to evaluate performance.

We generate workloads that contain 100 aggregation queries using the following query template:

Q_5:
SELECT AVG (x)
FROM G
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 selectivities of predicates in the different selectivity classes are as follows: $\{1*10^{-3}$, $5*10^{-3}$, $1*10^{-2}$, $5*10^{-2}$, $1*10^{-1}$, 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. Having three workloads allows us to have a test workload that is different from the training workload, which is important to evaluate generality.

The first experiment in this section explores the relationship between the number of unique prefixes recommended by a traditional design advisor and the number of queries

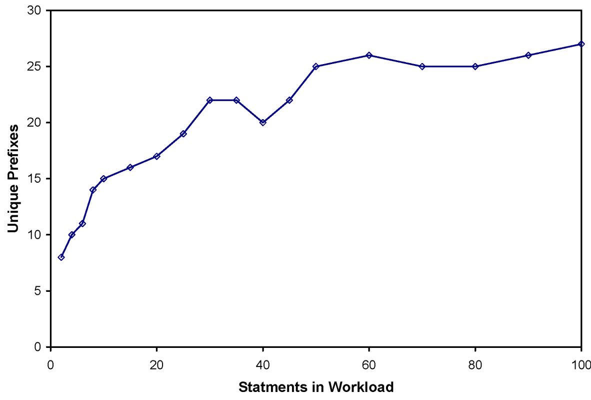

Figure 10: Unique prefixes - $4\times$ DB size

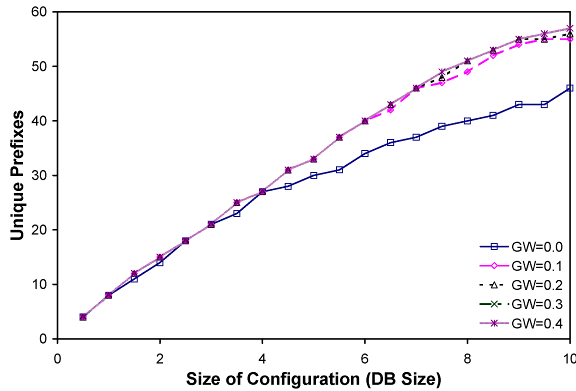

Figure 11: Unique prefixes - varying size constraint.

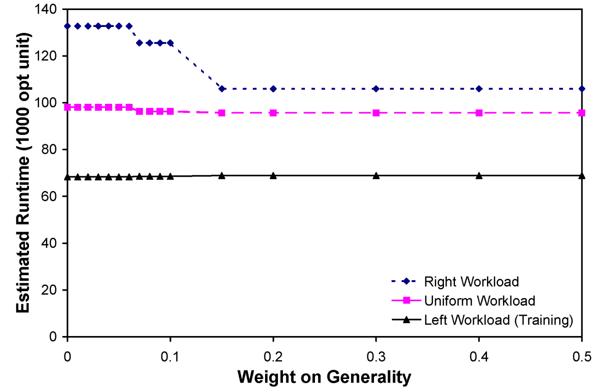

Figure 12: Evaluating generality - $3\times$ DB size

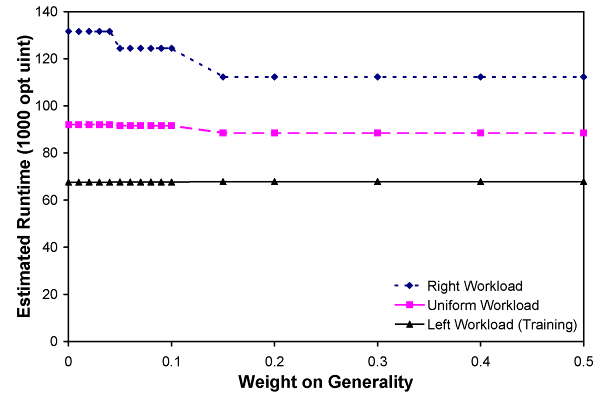

Figure 13: Evaluating generality - $4\times$ DB size

in the training workload. The goal is to show that a traditional design advisor makes more general recommendations as it sees more queries. Figures 9 and 10 show the number of unique prefixes in the configuration recommended by the MODA design advisor for a disk space budget of 3 times and 4 times the database size, respectively. In each figure, we vary the number of queries in the training workload from 2 to 100, using queries in the Left workload. Since we are modeling a traditional design advisor, we set the weight on the benefit metric to 1 and the weights on the risk and generality metrics to 0. We observe that the number of unique prefixes increases by increasing the number of queries in the training workload, which means that the design advisor becomes more general and less overtrained as it sees more queries. It also means that the more queries there are in the workload, the larger the number of unique prefixes that can be useful. This validates our goal of increasing the number of unique prefixes even for a small number of queries in the training workload. In the current experiment, the design advisor that saw, say, 20 queries will not be able to recommend the unique prefixes that are useful for the remaining 80 queries in the workload. The generality metric is aimed at enabling the design advisor to recommend such prefixes.

In our next experiment, we study the effectiveness of our generality metric in capturing the number of unique prefixes and helping the design advisor maximize it. Figure 11 shows the number of unique prefixes recommended by the MODA design advisor for the 100 queries in the Left workload. The figure shows values for five different settings of the weight on generality (GW), and for varying index configuration size. The weight on risk is set to 0. When the weight on generality is set to 0, MODA behaves like a traditional design advisor. As the weight on generality increases, MODA starts to maximize the number of unique index prefixes more rapidly than the traditional design advisor. This shows that the generality metric achieves its goal of maximizing number of unique prefixes.

In our final experiment, we demonstrate that the generality metric helps the design advisor benefit queries beyond those seen in the training workload. We train the design advisor with the Left workload, and we implement the configuration it recommends. We then test performance with the Left, Right, and Uniform workloads. Recall that each workload has 100 queries. We set the weight on risk metric, q_2, to 0, and we vary the weight on the generality metric, q_3. The weight on benefit is set to $q_1 = 1 - q_3$. Figures 12 and 13 show the results of this experiment for index configurations that are 3 and 4 times the database size, respectively. The figures show that increasing the weight on generality does not hurt the performance of the training workload (Left) but it improves the performance of the test workloads (Right and Uniform). Thus, we can see that the generality metric helps us achieve the goal of recommending robust physical designs that benefit not only the training workload, but also previously unseen test workloads.

8. CONCLUSION

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 paper, 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. The *Risk* metric quantifies confidence in the query optimizer costing of the benefit of an index to a query workload. The *Generality* metric quantifies the robustness of an index configuration to changes in the query workload. These two metrics can be combined with the expected benefit of the recommended physical design in a multi-objective design advisor. We have implemented such an advisor for PostgreSQL, and we show experimentally using this implementation that our approach does indeed result in physical design recommendations that are more robust.

For future work, it would be interesting to expand our current 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. It would also be interesting to investigate robustness in the context of other physical design decisions, such as recommending materialized views. Also, since many database systems are moving towards automatically collecting workload aware multi-column and join statistics, it would be interesting to see how much these statistics would reduce risk, and how to adapt the risk metric to systems that use these statistics. Finally, it would be interesting to integrate our notion of robustness into an on-line physical design tuner as a step towards truly zero administration database systems.

ACKNOWLEDGMENT

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

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

[2] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *Proc. SIGMOD*, 1999.

[3] A. Aboulnaga, P. J. Haas, S. Lightstone, G. M. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in DB2 UDB. In *Proc. VLDB*, 2004.

[4] S. Agrawal, S. Chaudhuri, L. Kollár, A. P. Marathe, V. R. Narasayya, and M. Syamala. Database tuning advisor for Microsoft SQL Server 2005. In *Proc. VLDB*, 2004.

[5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. VLDB*, 2000.

[6] S. Agrawal, E. Chu, and V. R. Narasayya. Automatic physical design tuning: Workload as a sequence. In *Proc. SIGMOD*, 2006.

[7] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *Proc. SIGMOD*, 2005.

[8] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proc. SIGMOD*, 2005.

[9] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. SIGMOD*, 2002.

[10] N. Bruno and S. Chaudhuri. Efficient creation of statistics over query expressions. In *Proc. ICDE*, 2003.

[11] N. Bruno and S. Chaudhuri. Conditional selectivity for statistics on query expressions. In *Proc. SIGMOD*, 2004.

[12] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proc. SIGMOD*, 2005.

[13] N. Bruno and S. Chaudhuri. Physical design refinement: The "merge-reduce" approach. In *Proc. EDBT*, 2006.

[14] N. Bruno and S. Chaudhuri. To tune or not to tune? A lightweight physical design alerter. In *Proc. VLDB*, 2006.

[15] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *Proc. ICDE*, 2007.

[16] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proc. SIGMOD*, 2001.

[17] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing SQL workloads. In *Proc. SIGMOD*, 2002.

[18] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. In *Proc. SIGMOD*, 1998.

[19] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proc. VLDB*, 1997.

[20] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zaït, and M. Ziauddin. Automatic SQL tuning in Oracle 10g. In *Proc. VLDB*, 2004.

[21] K. El Gebaly. Robustness in automatic physical database design. Master's thesis, University of Waterloo, 2007. Also available as University of Waterloo Computer Science Technical Report CS-2007-29.

[22] A. C. König and S. U. Nabar. Scalable exploration of physical database design. In *Proc. ICDE*, 2006.

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

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

[25] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proc. ICDE*, 2000.

[26] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *Proc. VLDB*, 2004.