CliffGuard: An Extended Report *

Barzan Mozafari Eugene Zhen Ye Goh Dong Young Yoon
University of Michigan, Ann Arbor
{mozafari,vanblaze,dyoon}@umich.edu

ABSTRACT

A fundamental problem in database systems is choosing *the best physical design*, i.e., a small set of auxiliary structures that enable the fastest execution of future queries. Almost all commercial databases come with designer tools that create a number of indices or materialized views (together comprising the *physical design*) that they exploit during query processing. Existing designers are what we call *nominal*; that is, they assume that their input parameters are precisely known and equal to some nominal values. For instance, since future workload is often not known *a priori*, it is common for these tools to optimize for past workloads in hopes that future queries and data will be similar. In practice, however, these parameters are often noisy or missing. Since nominal designers do not take the influence of such uncertainties into account, they find designs that are sub-optimal and remarkably brittle. Often, as soon as the future workload deviates from the past, their overall performance falls off a cliff. Thus, we propose a new type of database designer that is *robust* against parameter uncertainties, so that overall performance degrades more gracefully when future workloads deviate from the past. Users express their risk tolerance by deciding on how much nominal optimality they are willing to trade for attaining their desired level of robustness against uncertain situations. To the best of our knowledge, this paper is the first to adopt the recent breakthroughs in *robust optimization theory* to build a practical framework for solving one of the most fundamental problems in databases, replacing today's brittle designs with robust designs that guarantee a predictable and consistent performance.

1. INTRODUCTION

Database management systems are among the most critical software components in our world today. Many important applications across enterprise, science, and government depend on database technology to derive insight from their data and make timely decisions. To fulfill this crucial role, a database (or its administrator) must make many important decisions on how to provision and tune the system in order to deliver the best performance possible, such as which materialized views, indices, or samples to build. While these auxiliary structures can significantly improve performance, they also incur storage and maintenance overheads. In fact, most practical budgets only allow for building a handful of indices and a dozen materialized views out of an exponential number of possible structures. For instance, for a data-warehouse with 100 columns, there are at least $\Omega(3^{100})$ sorted projections to choose from (each column can be either absent, or in ascending/ descending order). Thus, a fundamental database problem is finding the best *physical design*; that is, finding a set of indices and/or materialized views that optimizes the performance of future queries.

Modern databases come with designer tools (a.k.a. auto-tuning tools) that take certain parameters of a target workload (e.g., queries, data distribution, and various cost estimates) as input, and then use different heuristics to search the design space and find an optimal design (e.g., a set of indices or materialized views) within their time and storage budgets. However, these designs are only optimal for the input parameters provided to the designer. Unfortunately, in practice, these parameters are subject to many sources of uncertainty, such as noisy environments, approximation errors (e.g., in the query optimizer's cost or cardinality estimates [16]), and missing or time-varying parameters. Most notably, since future queries are unknown, these tools usually optimize for past queries in *hopes* that future ones will be similar.

Existing designer tools (e.g., Index Tuning Wizard [10] and Tuning Advisor in Microsoft SQL Server [8, 32], Teradata's Index Wizard [26], IBM DB2's Design Advisor [89], Oracle's SQL Tuning Adviser [38], Vertica's DBD [57, 75, 83], and Parinda for Postgres [64]) do not take into account the *influence of such uncertainties* on the optimality of their design, and therefore, produce designs that are *sub-optimal* and *remarkably brittle*. We call all these existing designers ***nominal***. That is, all these tools assume that their input parameters are precisely known and equal to some nominal values. As a result, overall performance often plummets as soon as future workload deviates from the past (say, due to the arrival of new data or a shift in day-to-day queries). These dramatic performance decays are severely disruptive for time-critical applications. They also waste critical human and computational resources, as dissatisfied customers request vendor inspections, often resulting in re-tuning/re-designing the database to restore the required level of performance.

Our Goal — To overcome the shortcomings of nominal designers, we propose a new type of designers that are immune to parameter uncertainties as much as desired; that is, they are ***robust***. Our robust designer gives database administrators a *knob* to decide exactly how much nominal optimality to trade for a desired level of robustness. For instance, users may demand a set of optimal materialized views with an assurance that they must remain robust against change in their workload of up to 30%. A more conservative user may demand a higher degree of robustness, say 60%, at the expense of less nominal optimality. Robust designs are highly superior to nominal ones, as:

(a) Nominal designs are inherently brittle and subject to performance cliffs, while the performance of a robust design will degrade *more gracefully*.

(b) By taking uncertainties into account, robust designs can guard against worst-case scenarios, delivering a more consistent and predictable performance to time-sensitive applications.

(c) Given the highly non-linear and complex (and possibly non-convex) nature of database systems, a workload may have more than one optimal design. Thus, it is completely conceivable that a robust design may be nominally optimal as well (see [21, 22] for such examples in other domains).

*A 16-page version of this manuscript has appeared in proceedings of ACM SIGMOD, 2015 [70]

(d) A robust design can significantly reduce operational costs by requiring less frequent database re-designs.

Previous Approaches — There has been some pioneering work on incorporating parameter uncertainties in databases [16, 31, 37, 42, 66, 74]. These techniques are specific to run-time query optimization and do not easily extend to physical designs. Other heuristics have been proposed for improving physical designs through workload compression (i.e., omitting workload details) [30, 55] or modifying the query optimizer to return richer statistics [44]. Unfortunately, these approaches are not principled and thus do not necessarily guarantee robustness. (In Section 6.4, we compare against commercial databases that use such heuristics.)

To avoid these limitations, adaptive indexing schemes such as Database Cracking [48, 52] take the other extreme by completely ignoring the past workload in deciding which indices to build; instead of an *offline* design, they incrementally create and refine indices as queries arrive, *on demand*. However, even these techniques need to decide which subsets of columns to build an incremental index on.[1] Instead of completely relying on past workloads or abandoning the offline physical design, in this paper we present a principled framework for directly maximizing robustness, which enables users to decide on the extent to which they want to rely on past information, and the extent of uncertainty they want to be robust against. (We discuss the merits of previous work in Section 7.)

Our Approach — Recent breakthroughs in Operations Research on robust optimization (RO) theory have created new hopes for achieving robustness and optimality in a principled and tractable fashion [21, 22, 36, 88]. In this paper, we present the first attempt at applying RO theory to building a practical framework for solving one of the most fundamental problems in databases, namely finding the best physical design. In particular, we study the effects of workload changes on query latency. Since OLTP workloads tend to be more predictable (e.g., transactions are often instances of a few templates [68, 69]), we focus on OLAP workloads where exploratory and ad-hoc queries are quite common. Developing this robust framework is a departure from the traditional way of designing and tuning databases: from today's brittle designs to a principled world of robust designs that guarantee a predictable and consistent performance.

RO Theory — The field of RO has taken many strides over the past decade [21]. In particular, the seminal work of Bertsimas et al. [22] has been successfully applied to a number of drastically different domains, from nano-photonic design of telescopes [22] to thin-film manufacturing [24] and system-on-chip architectures [71]. To the best of our knowledge, developing a principled framework for applying RO theory to physical design problems is the first application of these techniques in a database context, which involves a number of unique challenges not previously faced in any of these other applications of RO theory (discussed in Section 4.2).

A common misconception about the RO framework is that it requires knowledge of the extent of uncertainty, e.g., in our case, an upper bound on how much the future workload will deviate from the past one.[2] To the contrary, the power of the RO formulation is that it allows users to freely request any degree of robustness that they wish, say Γ, *purely* based on their own risk tolerance and preferences [19, 23]. Regardless of whether the actual amount of uncertainty exceeds or stays lower than Γ, the RO framework guarantees will remain valid; that is, the delivered design is promised to remain optimal as long as the uncertainty remains below the user-requested threshold Γ, and beyond that (i.e., if uncertainty exceeds Γ) is in accordance to user's accepted degree of risk [23]. In other words, the beauty of RO theory is that it provides a framework for expressing and delivering reliability guarantees by decoupling them from the actual uncertainty in the environment (here, the future workload).

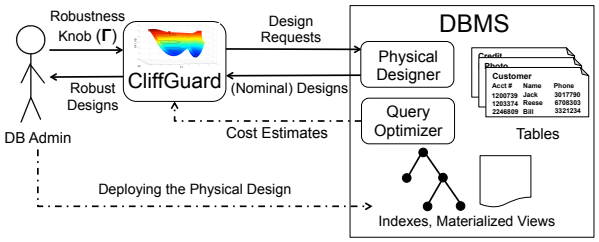


Figure 1: The `CliffGuard` architecture.

Contributions — In this paper, we make these contributions:

- We formulate the problem of robust physical design using RO theory (Section 3).
- We design a principled algorithm, called `CliffGuard`, by adapting the state-of-the-art framework for solving non-convex RO problems. `CliffGuard`'s design is generic and can potentially work with any existing designers and databases without modifying their internals (Section 4).
- We implement and evaluate `CliffGuard` using two major commercial databases (HP Vertica and DBMS-X[3]) on two synthetic workloads as well as a real workload of 430+K OLAP queries issued by one of Vertica's major customers over a 1-year period (Section 6).

In summary, compared to Vertica's state-of-the-art commercial designer [57, 83], our robust designer reduces the average and maximum latency of queries on average by $7\times$ and $18\times$ (and up to $14\times$ and $40\times$), respectively. Similarly, `CliffGuard` improves over DBMS-X by 3–$5\times$. CliffGuard is currently available as an open-source, third-party tool [1].

2. SYSTEM OVERVIEW

Physical Database Designs — A physical design in a database is a set of auxiliary structures, often built *offline*, which are used to speed up future queries as they arrive. The type of auxiliary structures used often depend on the specific database architecture. Most databases use both materialized views and indices in their physical designs. Materialized views are typically more common in analytical workloads. Approximate databases use small samples of the data (rather than its entirety) to speed up query processing at the cost of accuracy [3, 4, 5, 29, 85, 86]. Physical designs in these systems consist of different types of samples (e.g., stratified on different columns [6, 29]). Some modern columnar databases, such as Vertica [78, 83], build a number of column *projections*, each sorted differently. Instead of traditional indices, Vertica chooses a projection with the appropriate sort order (depending on the columns in

[1] Moreover, on-demand and continuous physical re-organizations are not acceptable in many applications, which is why nearly all commercial databases still rely on their offline designers.

[2] This misconception is caused by differing terminology used in other disciplines, such as mechanical engineering (ME) where "robust optimization" refers to a different type of optimization which requires some knowledge of the uncertainty of the physical environment [39]. The Operations Research notion of RO used in this paper is called *reliability optimization* in the ME literature [80].

[3] DBMS-X is a major database system, which we cannot reveal due to the vendor's restrictions on publishing performance results.

the query) in order to locate relevant tuples quickly. In all these examples, the space of these auxiliary structures is extremely large if not infinite, e.g., there are $O(2^N \cdot N!)$ possible projections or indices for a table of N columns (i.e., different subsets and orders of columns). Thus, the physical design problem is choosing a small number of these structures using a fixed budget (in terms of time, space, or maintenance overhead) such that the overall performance is optimized for a target workload.

Design Principles — A major goal in the design of our `CliffGuard` algorithm is compatibility with almost any existing database in order to facilitate its adoption in the commercial world. Thus, we have made two key decisions in our design. First, `CliffGuard` should operate alongside an existing (nominal) designer rather than replacing it. Despite their lack of robustness, existing designers are highly sophisticated tools hand-tuned over the years to find the best physical designs efficiently, given their input parameters. Because of this heavy investment, most vendors are reluctant to abandon these tools completely. However, some vendors have expressed interest in `CliffGuard` as long as it can operate alongside their existing designer and *improve* its output. Second, `CliffGuard` is designed to treat existing designers as a *black-box* (i.e., without modifying their internal implementations). This is to conform to the proprietary nature of commercial designers and also to widen the applicability of `CliffGuard` to different databases. By delegating the nominal designs to existing designers, `CliffGuard` remains a genetic framework agnostic to the specific details of the design objects (e.g., they can be materialized views, samples, indices, or projections).

These design principles have already allowed us to evaluate `CliffGuard` for two database products with drastically different design problems (i.e., Vertica and DBMS-X). Without requiring any changes to their internal implementations, `CliffGuard` significantly improves on the sophisticated designers of these leading databases (see Section 6). Thus, we believe that `CliffGuard` can be easily used to speed up other database systems as well.

Architecture — Figure 1 depicts the high-level workflow of how `CliffGuard` is to be used alongside a database system. The database administrator states her desired degree of robustness Γ to `CliffGuard`, which is located outside the DBMS. `CliffGuard` in turn invokes the existing physical designer via its public API. After evaluating the output (nominal) design sent back from the existing designer, `CliffGuard` may decide to manipulate the existing designer's output by merely modifying some of its input parameters (in a principled manner) and invoking its API again. `CliffGuard` repeats this process, until it is satisfied with the robustness of the design produced by the nominal designer. The final (robust) design is then sent back to the administrator, who may decide to deploy it in the DBMS.

3. PROBLEM FORMULATION

In this section, we present a simple but powerful formulation of robustness in the context of physical database design. This formulation will allow us to employ recently proposed ideas in the theory of robust optimization (RO) and develop a principled and effective algorithm for finding robust database designs, which will be presented in Section 4. First, we define some notations.

Notations — For a given database, the **design space** $\mathcal{S}$ is the set of all possible structures of interest, such as indices on different subsets of columns, materialized views, different samples of the data, or a combination of these. For example, Vertica's designer [78, 83] materializes a number of *projections*, each sorted differently:

```
CREATE PROJECTION   projection_name
  AS SELECT   col1, col2, ..., colN
  FROM   anchor_table
  ORDER BY   col1', col2', ..., colK';
```

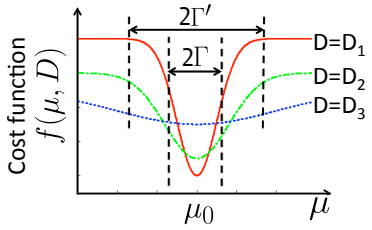


Figure 2: Design D_1 is nominally optimal at μ_0 while designs D_2 and D_3 are robust against an uncertainty of $\pm\Gamma$ and $\pm\Gamma'$ in our parameter μ_0, respectively.

Here, $\mathcal{S}$ is extremely large due to the exponential number of possible projections. Similarly, for decisions about building materialized views or (secondary) indices, $\mathcal{S}$ will contain all such possible structures. Existing database designers solve the following optimization problem (or aim to[4]):

$$D^{nom} = \mathbb{D}(W_0, B) = \underset{D \subseteq \mathcal{S},\ price(D) \leq B}{ArgMin} f(W_0, D) \qquad (1)$$

where W_0 is the target **workload** (e.g., the set of user queries), B is a given **budget** (in terms of storage or maintenance overhead), $\mathbb{D}$ is a **nominal designer** that takes a workload and budget as input parameters, $price(D)$ is the **price** of choosing D (e.g., the total size of the projections in D), and $f(W_0, D)$ is our **cost function** for executing workload W_0 using design D (e.g., f can be the query latency). We call such designs **nominal** as they are optimal for the nominal value of the given parameters (e.g., the target workload). All existing designers [10, 38, 64, 75, 89] are *nominal*: they either minimize the expression above directly, or follow other heuristics aimed at approximate minimization. Despite several heuristics to avoid over-fitting a given workload (e.g., omitting query details [30, 55]), nominal designers suffer from many shortcomings in practice; see Sections 1 and 6.4.

Robust Designs — This paper's goal is finding designs that are robust against *worst-case* scenarios that can arise from uncertain situations. This concept of *robustness* can be illustrated using the toy example of Figure 2, which features a design space with only three possible designs and a toy workload that is represented by a single real-value parameter μ. When our current estimate of μ is μ_0, a nominal designer will pick design D_1 since it minimizes the cost at μ_0. But if we want a design that remains optimal even if our parameter changes by up to Γ, then a robust designer will pick design D_2 instead of D_1, even though the latter has a lower cost at μ_0. This is because the *worst-case* cost of D_2 over the $[\mu_0 - \Gamma, \mu_0 + \Gamma]$ is lower than that of D_1; that is, D_2 is robust against uncertainty of up to Γ. Similarly, if we decide to guard against a still greater degree of uncertainty, say for an estimation error as high as $\Gamma' > \Gamma$, a robust designer would this time pick D_3 instead of D_2, as the former has a lower worst-case cost in $[\mu_0 - \Gamma', \mu_0 + \Gamma']$ than the other designs.

Formally, a robust design D^{rob} can be defined as:

$$D^{rob} = \tilde{\mathbb{D}}(W_0, B, \mathcal{U}) = \underset{D \subseteq \mathcal{S},\ price(D) \leq B}{ArgMin} \ \underset{W \in \mathcal{U}(W_0)}{Max} f(W, D)$$

where $\mathcal{U}(W_0)$ defines an uncertainty region around our target workload W_0. Here, $\tilde{\mathbb{D}}$ is a robust designer that will search for a design that minimizes the cost function regardless of where the target

[4]Existing designers often use heuristics or greedy strategies [65], which lead to *approximations* of the nominal optima.

workload lands in this uncertainty region. In other words, this MiniMax formulation of robustness defines a robust design as one with the **best worst-case performance**.

Although the uncertainty region $\mathcal{U}(W_0)$ does not have to be circular, for ease of presentation in this paper we always define $\mathcal{U}(W_0)$ as a circular region of radius $\Gamma \geq 0$ centered at W_0, which we call the Γ**-neighborhood** of W_0. For instance, the Γ-neighborhood will be an interval when $W_0 \in \mathbb{R}$ (see Figure 2) and a circle when $\Gamma \in \mathbb{R}^2$. Since database workloads are not easily represented as real numbers, we need to use a distance function to define the Γ-neighborhood of a database workload W_0, namely:

$$D^{rob} = \tilde{\mathbb{D}}(W_0, B, \Gamma) = \underset{D \subseteq \mathcal{S},\ price(D) \leq B}{ArgMin} \ \underset{\delta(W, W_0) \leq \Gamma}{Max} \ f(W, D) \tag{2}$$

Here, $\delta(.)$ is a user-defined distance function that takes a pair of workloads and returns a non-negative real number as their distance. Formulation (2) allows users to express their desired level of robustness by choosing the value of $\Gamma \geq 0$, where the larger the Γ, the more robust their design is. Note that a nominal design is a special case of a robust design where $\Gamma = 0$. In the rest of this paper, we will not explicitly mention the $price(D) \leq B$ constraint in our notations, but it will always be implied in both nominal and robust designs. (Appropriate choices of Γ and δ are discussed later in this section and in Section 5, respectively.)

We have chosen this formulation of robustness for several reasons. First, (2) is the standard notion of *robust optimization* in Operations Research [21], which enables us to adopt recently proposed RO algorithms that are highly tractable and effective (see Section 4). Second, formulation (2) requires no knowledge of the uncertainty distribution. (We will discuss the role of Γ shortly.) This is in contrast to an alternative formulation seeking to minimize the *expected* or 99%-quantile latency (rather than its worst-case). However, such a formulation would be a different type of optimization, called *stochastic optimization* [36], requiring a probability distribution on every possible workload (which is impossible to obtain in most real-world scenarios). Finally, despite their MiniMax (i.e., worst-case) formulation, the RO solutions may not be overly conservative; surprisingly, they are often similar to those produced by stochastic optimization, which is substantially less tractable (see [21]). In fact, in our database context, RO solutions are also nominally superior, thanks to databases' non-convex cost functions and complex design spaces (see Section 6.4).

A Knob for Robustness — As mentioned in Section 1, the role of Γ in RO formulation (2) is sometimes misunderstood to be an upper bound on the degree of uncertainty, i.e., Γ should be chosen such that the future workload W will lie in W_0's Γ-neighborhood. To the contrary, the beauty of formulation (2) is that it allows users to choose any Γ value based purely on their own business needs and risk tolerance, regardless of the actual amount of uncertainty in the future. In other words, Γ is *not* an upper bound on the actual uncertainty in the environment, but rather the amount of actual uncertainty that the user decides to guard against. This is a subtle but important distinction, because robustness comes at the price of reduced nominal optimality. In the example of Figure 2, D_2 is robust against a greater degree of uncertainty than D_2 but is nominally more expensive at $\mu = \mu_0$. Therefore, it is important to interpret Γ as a *robustness knob* and not a prediction of future uncertainty.

The choice of Γ depends completely on the end users' risk tolerance and is not the focus of this paper. Our paper's contribution is a framework that will deliver a design that guarantees the requested level of robustness for any value of Γ chosen by the user. For instance, a user may take the simplest approach and use the sequence of workload changes over the past N windows of queries, say

$$\delta(W_0, W_1), \delta(W_1, W_2), \cdots, \delta(W_{N-1}, W_N)$$

and take their average, max, or $k\times$max (for some constant $k>1$) as a reasonable choice of Γ when finding a robust design for W_{N+1} using W_N. Alternatively, a user may employ more sophisticated techniques (e.g., timeseries forecasting [28]) to obtain a more accurate prediction for $\delta(W_N, W_N + 1)$. Regardless of the strategy, the actual uncertainty can always exceed a user's predictions. However, this problem is no different from any other provisioning problem. For instance, many customers provision their database resources according to, say, $3\times$ their current peak load. This means that according to their business needs, they accept the risk of their future workload suddenly increasing by $4\times$. This is analogous to the user's choice of Γ here. Also, note that even if users magically knew the exact value of $\delta(W_N, W_{N+1})$ in advance, the existing nominal designers' performance would remain the same since they have no mechanism for incorporating a bounded uncertainty into their analysis. (A nominal designer would only perform better if we knew the actual W_{N+1} and not just its distance from W_N.) As previously explained, while our proposed designer does not *require* any prior knowledge of the uncertainty in order to deliver the user's robustness requirements, it can naturally incorporate additional of the future workload if made available by the user.

In Section 6.5, we study the effects of different Γ choices and show that, in practice, our algorithm performs no worse than the nominal designer even when presented with poor (i.e., extremely low or extremely high) Γ choices.

4. OUR ALGORITHM

In the previous section, we provided the RO formulation of the physical design for databases. Over the past decade, there have been many advances in the theory of RO for solving problems with similar formulations as (2), for many different classes of cost functions and uncertainty sets (see [21] for a recent survey). Here, the most relevant to our problem is the seminal work of Bertsimas et al. [22], hereon referred to as the BNT algorithm. Unlike most RO algorithms, BNT does not require the cost function to have a closed-form. This makes BNT an ideal match for our database context: our cost function is often the query latency, which does not have an explicit closed-form, i.e., latency can only be *measured* by executing the query itself or *approximated* using the query optimizer's cost estimates. BNT's second strength is that it does not require convexity: BNT guarantees a global robust solution when the cost function is convex, and convergence to a local robust solution even when it is not convex. Given the complex nature of modern databases, establishing convexity for query latencies can be difficult (e.g., in some situations, additional load can reduce latency by improving the cache hit rate [68]).[5]

First, we provide background on the BNT framework in Section 4.1. Then, in Section 4.2, we identify the unique challenges that arise in applying BNT to database problems. Finally, in Section 4.3, we propose our BNT-based `CliffGuard` algorithm, which overcomes these challenges.

4.1 The BNT Algorithm

In this section, we offer a geometric interpretation of the BNT algorithm for an easier understanding of the main ideas behind the

[5]When the cost function is non-convex, the output of existing nominal designers is also only *locally* optimal. Thus, even in these cases, finding a local robust optimum is still a worthwhile endeavor.

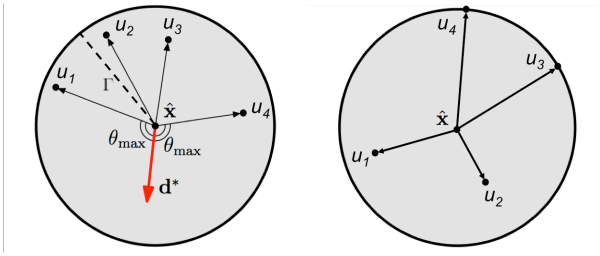


Figure 3: (a) A descent direction d^* is one that moves away from *all* the worst-neighbors ($\theta_{max} \geq 90°$); (b) here, due to the location of the worst-neighbors, no descent direction exists.

algorithm. (Interested readers can find a more formal discussion of the algorithm in the Operations Research literature [22].)

We use Figure 3(a) to illustrate how the BNT algorithm works. Here, imagine a simplified world in which each decision is a 2-dimensional point in Euclidean space. Since the environment is noisy or unpredictable, the user demands a decision x^* that comes with some *reliability* guarantees. For example, instead of asking for a decision x^* that simply minimizes $f(x)$, the user requires an x^* whose worst-case cost is minimized for arbitrary noise vectors Δx within a radius of Γ, namely:

$$x^* = ArgMin_{x} \; Max_{||\Delta x||_2 \leq \Gamma} \; f(x + \Delta x) \tag{3}$$

Here, $||\Delta x||_2$ is the length (L_2-norm) of the noise vectors. This means that the user expresses his/her reliability requirement as an uncertainty region, here a circle of radius Γ, and demands that our $f(x^* + \Delta x)$ still be minimized no matter where the noisy environment moves our initial decision within this region. This uncertainty region (i.e., the Γ-neighborhood) is shown as a shaded disc in Figure 3(a).

To meet the user's reliability requirement, the BNT algorithm takes a starting point, say $\hat{x}$, and performs a number of iterations as follows. In each iteration, BNT first identifies all the points within the Γ-neighborhood of $\hat{x}$ that have the highest cost, called the *worst-neighbors* of $\hat{x}$. In Figure 3(a), there are four worst-neighbors, shown as $u_1, \cdots, u_4$. Let $\Delta x_1, \cdots, \Delta x_4$ be the vectors that connect $\hat{x}$ to each of these worst-neighbors, namely $u_i = \hat{x} + \Delta x_i$ for i=1, 2, 3, 4.

Once the worst-neighbors of $\hat{x}$ are identified, the BNT algorithm finds a direction that moves away from all of them. This direction is called the *descent direction*. In our geometric interpretation, a descent direction $\vec{d}^*$ is one that maximizes the angle θ in Figure 3(a) by halving the reflex angle between the vectors connecting $\hat{x}$ to u_1 and u_4. The BNT algorithm then takes a small step along this descent direction to reach a new decision point, say $\hat{x}'$, which will be at a greater distance from all of the worst-neighbors of $\hat{x}$. The algorithm repeats this process by looking for the new worst-neighbors in the Γ-neighborhood of $\hat{x}'$. (Bertsimas et al. prove that taking an appropriately-sized step along the descent direction reduces the worst-case cost at each iteration [22].) The algorithm ends (i.e., a robust solution is found) when no descent direction can be found. Figure 3(b) illustrates this situation, as any direction of movement within the Γ-neighborhood will bring the solution closer to at least one of the worst-neighbors. (Bertsimas et al. prove that this situation can only happen when we reach a local robust minimum, which will also be a global robust minimum when the cost function is convex.)

To visually demonstrate this convergence, we again use a geometric interpretation of the algorithm, as depicted in Figure 4.1.

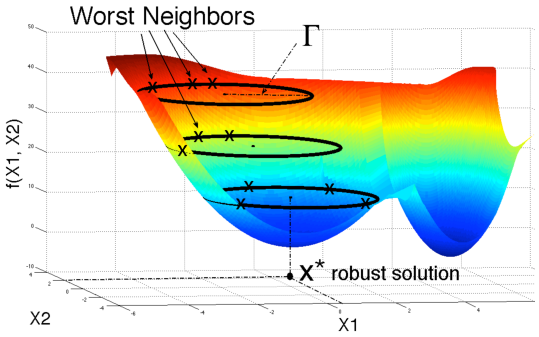


Figure 4: Geometric interpretation of the iterations in BNT.

In this figure, the decision space consists of two-dimensional real vectors $(x_1, x_2) \in \mathbb{R}^2$ and the $f(x_1, x_2)$ surface corresponds to the cost of different points in this decision space. Here, the Γ-neighborhood in each iteration of BNT is shown as a transparent disc of radius Γ. Geometrically, to move away from the worst-neighbors at each step is equivalent to sliding down this disc along the steepest direction such that the disc always remains within the cost surface and parallel to the (x_1, x_2) plane. The algorithm ends when this disc's boundary touches the cost surface and cannot be sliced down any further without breaking through the cost surface—this is the case with the bottom-most disc in Figure 4.1; when this condition is met, the center of this disc represents a locally robust solution of the problem (marked as x^*) and its worst-neighbors lie on the boundary of its disc (marked as $\times$). The goal of BNT is to quickly find such discs and converge to the locally robust optimum.

The pseudo of BNT is presented in Algorithm 1. Here, x_k is the current decision at the k'th iteration. As explained above, each iteration consists of two main steps: finding the worst-neighbors (neighborhood exploration, Line 5) and moving away from those neighbors if possible (local robust move, Lines 7–16). In Section 4.2, we discuss some of the challenges posed by these steps when applied to physical design problems in databases.

Theoretical Guarantees — When $f(x)$ is continuously differentiable with a bounded set of minimum points, Bertsimas et al. [22] show that their algorithm converges to the local optimum of the robust optimization problem (3), as long as the steps sizes t_k (Line 14 in Algorithm 1) are chosen such that $t_k > 0$, $\lim_{k\to\infty} t_k = 0$, and $\sum_{k=1}^{\infty} t_k = \infty$. For convex cost surfaces, this solution is also the global optimum. (With non-convex surfaces, BNT needs to be repeated from different starting points to find multiple local optima and choose one that is more globally optimal.[6])

4.2 Challenges of Applying BNT to Database Problems

As mentioned earlier, since BNT does not require a closed-form cost function (or even convexity), it presents itself as the most appropriate technique in the RO literature for solving our physical design problems, especially since we want to avoid modifying the internals of the existing designers (due to their proprietary nature, see Section 2). However, BNT still hinges on certain key assumptions that prevent it from being directly applicable to our design problem. Next, we discuss each of these requirements and the unique challenges that they pose in a database context.

Proper distance metric — BNT requires a *proper* distance metric over the decision space, i.e., one that is symmetric and satisfies the triangle property. E.g., the L2-norm $||x||_2$ is a proper distance over

[6]When $f(x)$ is non-convex, the output of existing designers is also a local optimum. Thus, even in this case, finding local robust optima is still preferable (to a local nominal optimum).

Algorithm 1: Generic robust optimization via gradient descent.

Inputs: Γ: the radius of the uncertainty region,
$f(x)$: the cost of design x
Output: x^*: a robust design, i.e.,
$x^* = \underset{x}{ArgMin} \underset{||\Delta x||_2 \leq \Gamma}{Max} f(x + \Delta x)$

1 $x_1 \leftarrow$ pick an arbitrary vector **// the initial decision**
2 $k \leftarrow 1$ **// k is the number of iterations so far**
3 **while** *true* **do**
// Neighborhood Exploration:
5 $U \leftarrow$ Find the set of worst-neighbors of x_k within its Γ-neighborhood
// Robust Local Move:
7 $\vec{\mathbf{d}}^* \leftarrow$ FindDescentDirection(x_k, U)
// See Fig 3(a) for FindDescentDirection**'s geometric intuition and Appendix B for its formal definition**
9 **if** there is no such direction $\vec{\mathbf{d}}^*$ pointing away from all $u \in U$ **then**
11 $x^* \leftarrow x_k$ **// found a local robust solution**
12 **return** x^*
else
14 $t_k \leftarrow$ choose an appropriate step size
15 $x_{k+1} \leftarrow x_k + t_k \cdot \vec{\mathbf{d}}^*$ **// move along the descent direction**
16 $k \leftarrow k + 1$ **// go to next iteration**
end
end

the m-dimensional Euclidean space, since $||x_1 - x_2||_2 + ||x_2 - x_3||_2 \geq ||x_1 - x_3||_2 = ||x_3 - x_1||_2$ for any $x_1, x_2, x_3 \in \mathbb{R}^m$.

Challenge C1. To define an analogous notion of uncertainty in a database context, we need to have a distance metric $\delta(W_1, W_2)$ for any two sets of SQL queries, say W_1 and W_2, in order to express the uncertainty set of an existing workload W_0 as $\{W \mid \delta(W_0, W) \leq \Gamma\}$. Note that δ must be symmetric, triangular, and also capable of capturing the user's notion of a *workload change*. To the best of our knowledge, such a distance metric does not currently exist for database workloads.[7]

Finding the worst-neighbors — BNT relies on our ability to find the worst-neighbors U (Algorithm 1, Line 5) in each iteration, which equates to finding *all* global maxima of the following optimization problem:

$$\underset{||\Delta x||_2 \leq \Gamma}{ArgMax} \quad f(x_k + \Delta x) \tag{4}$$

In other words, the worst-neighbors are defined as:

$$U = \{x_k + \Delta x \mid f(x_k + \Delta x) = g(x_k), \ ||\Delta x||_2 \leq \Gamma\}$$

where $g(x)$ represents our worst-case cost function, defined as:

$$g(x) = \underset{||\Delta x||_2 \leq \Gamma}{Max} \quad f(x + \Delta x)$$

When $g(x)$ is differentiable, finding its global maxima is straightforward, as one can simply take its derivative and solve the following equation:

$$g'(x) = 0 \tag{5}$$

All previous applications of the BNT framework have either involved a closed form cost function $f(x)$ with a differentiable worst-case cost function $g(x)$, where the worst-neighbors can be found by solving (5) (e.g., in industrial engineering [21] or chip design [71]), or a black-box cost function guaranteed to be continuously differentiable (e.g., in designing nano-photonic telescopes [22]).

Challenge C2. Unfortunately, most cost functions of interest in databases are not closed-form, differentiable, or even continuous. For instance, when f is the query latency, it does not have a closed-form; it is measured either via actual execution or by consulting the query optimizer's cost estimates. Also, even a small modification in the design or the query can cause a drastically different latency, e.g., when a query references a column that is omitted from a materialized view.

Finding a descent direction — BNT relies on our ability to efficiently find the (steepest) descent direction via a second-order cone program (SOCP) (see Appendix B), which requires a continuous domain.[8]

Challenge C3. We cannot use the same SOCP formulation because the space of physical designs is not continuous. A physical design, say a set of projections, can be easily encoded as a binary vector. For instance, each projection can be represented as a vector in $\{0, 1\}^m$ where the i'th coordinate represents the presence or absence of the i'th column in the database. Different column-orders and a set of such structures can also be encoded using more dimensions. However, this and other possible encodings of a database design are inherently discrete. For instance, one cannot construct a conventional projection with only 0.3 of a column—a column is either included in the projection or not.

Moving along a descent direction — BNT assumes that the decision space (i.e., the domain of x) is continuous and hence, moving along a descent direction is trivial (Algorithm 1, Line 15). In other words, if x_k is a valid decision, then $x_k + t_k \cdot \vec{\mathbf{d}}^*$ is also a valid decision for any given $\mathbf{d}^*$ and $t_k > 0$.

Challenge C4. Even when a descent direction is found in the database design space, moving along that direction does not have any database equivalence. In other words, even when our vectors x_k and $\vec{\mathbf{d}}^*$ correspond to legitimate physical designs, $x_k + t_k \cdot \vec{\mathbf{d}}^*$ may no longer be meaningful since it may not correspond to any legitimate design, e.g., it may involve fractional coordinates for some of the columns depending on the value of t_k. Thus, we need to establish a different notion of *moving along a descent direction* for database designs.

In summary, in order to use BNT's principled framework, we need to develop analogous techniques in our database context for expressing distance and finding the worst-neighbors; we also need to define equivalent notions for finding and moving along a descent direction. Next, we explain how our `CliffGuard` algorithm overcomes challenges C1–C4 and uses BNT's framework to find robust physical database designs.

4.3 Our Algorithm: CliffGuard

In this section, we propose our novel algorithm, called `CliffGuard`, which builds upon BNT's principled framework by tailoring it to the problem of physical database design.

Before presenting our algorithm, we need to clarify a few notional differences. Unlike BNT, where the cost function $f(x)$ takes a single parameter x, the cost in `CliffGuard` is denoted as a two-parameter function $f(W, D)$ where W is a given workload and D is a given physical design. In other words, each point x in our space is a pair of elements (W, D). However, unlike BNT where vector

[7] While workload drift is well-observed in the database community [49, 50, 76], quantifying it has received surprisingly little attention.

[8] SOCPs can be solved efficiently via interior point methods [25].

x can be updated in its entirety, in CliffGuard (or any database designer) we only update the design element D; this is because the database designer can propose a new physical design to the user, but cannot impose a new workload on her as a means to improve robustness.

Algorithm 2 presents the pseudo code for CliffGuard. Like Algorithm 1, Algorithm 2 iteratively explores a neighborhood to find the worst-neighbors, then moves farther away from these neighbors in each iteration using an appropriate direction and step size. However, to apply these ideas in a database context (i.e., addressing challenges C1–C4 from Section 4.2), Algorithm 2 differs from Algorithm 1 in the following important ways.

Initialization (Algorithm 2, Lines 1–2) — CliffGuard starts by invoking the existing designer $\mathbb{D}$ to find a nominal design D for the initial workload W_0. (Later, D will be repeatedly replaced by designs that are more robust.) CliffGuard also creates a finite set of perturbed workloads $P = \{W_1, \cdots, W_n\}$ by sampling the workload space in the Γ-neighborhood of W_0. In other words, given a distance metric δ, we find n workloads $W_1, \cdots, W_n$ such that $\delta(W_i, W_0) \leq \Gamma$ for $i = 1, 2, \cdots, n$. (Section 5 discusses how to define δ for database workloads, how to choose n, and how to sample the workload space efficiently.) Next, as in BNT, CliffGuard starts an iterative search with a neighborhood exploration and a robust local move in each iteration.

Neighborhood Exploration (Algorithm 2, Line 6) — To find the worst-neighbors, in CliffGuard we need to also take the current design D into account (i.e., the set of worst-case neighbors of W_0 will depend on the physical design that we choose). Given that we cannot rely on the differentiability (or even continuity) of our worst-case cost function (Challenge C2), we use the worst-case costs on our sampled workloads P a proxy; instead of solving

$$\underset{\delta(W,W_0)\leq\Gamma}{Max} f(W, D) \tag{6}$$

we solve

$$\underset{W\in P}{Max} f(W, D) \tag{7}$$

Note that (7) cannot provide an unbiased approximation for (6) simply because P is a finite sample, and finite samples lead to biased estimates for extreme statistics such as min and max [82]. Thus, we do not rely on the nominal value of (7) to evaluate the quality of our design. Rather, we use the solutions to (7) as a proxy to guide our search in moving away from highly (though not necessarily the most) expensive neighbors. In our implementation, we further mitigate this sampling bias by loosening our selection criterion to include all neighbors that have a high-enough cost (e.g., top-K or top 20%) instead of only those that have the maximum cost. To implement this step, we simply enumerate each workload in P and measure its latency on the given design.

Robust Local Move (Algorithm 2, Lines 8–15) — To find equivalent database notions for finding and moving along a descent direction (C3 and C4), we use the following idea. The ultimate goal of finding and moving along a descent direction is to reduce the worst-case cost of the current design. In CliffGuard, we can achieve this goal directly by *manipulating* the existing designer by feeding it a mixture of the existing workload and its worst-neighbors as a single workload.[9] The intuition is that since nominal designers (by definition) produce designs that minimize the cost of their input workload, the cost of our previous worst-neighbors will no longer be as high, which is equivalent to moving our design farther away from those worst-neighbors. The questions then are (i) how do we mix these workloads, and (ii) what if the designer's output leads to a higher worst-case cost?

The answer to question (i) is a weighted union, where we take the union of all the queries in the original workload as well as those in the worst-neighbors, after weighting the latter queries according to a scaling factor α, their individual frequencies of occurrence in their workload, and their latencies against the current design. Taking latencies and frequencies into account encourages the nominal designer to seek designs that reduce the cost of more expensive and/or popular queries. Scaling factor α, which serves the same purpose as step-size in BNT, allows CliffGuard to control the distance of movement away from the worst-neighbors.

We also need to address question (ii) because unlike BNT, where the step size t_k could be computed to ensure a reduction in the worst cost, here our α factor may in fact lead to a worse design (e.g., by moving too far from the original workload). To solve this problem, CliffGuard dynamically adjusts the step-size using a common technique called *backtracking line search* [25], similar to a binary-search. Each time the algorithm succeeds in moving away from the worst-neighbors, we consider a larger step size (by a factor $\lambda_{success} > 1$) to speed up the search towards the robust solution, and each time we fail, we reduce the step size (by a factor $0 < \lambda_{failure} < 1$) as we may have moved past the robust solution (hence observing a higher worst-case cost).

Termination (Algorithm 2, Lines 17–20) — We repeat this process until we find a local robust optimum (or reach the maximum number of steps, when under a time constraint).

5. EXPRESSING ROBUSTNESS GUARANTEES

In this section, we define a database-specific distance metric δ so that users can express their robustness requirements by specifying a Γ-neighborhood (as an uncertainty set, described in Section 3) around a given workload W_0, and demanding that their design must be robust for any future workload W as long as $\delta(W_0, W) \leq \Gamma$. Thus, users can demand arbitrary degrees of robustness according to their performance requirements. For mission-critical applications more sensitive to sudden performance drops, users can be more conservative (specifying a larger Γ). At the other extreme, users expecting no change (or less sensitive to it) can fall back to the nominal case ($\Gamma = 0$).

A distance metric δ must satisfy the following criteria to be effectively used in our BNT-based framework (Appendix D provides the intuition behind these criteria):

(a) *Soundness*, which requires that the smaller the distance $\delta(W_1, W_2)$, the better the performance of W_2 on W_1's nominally optimal design. Formally, we call a distance metric *sound* if it satisfies:

$$\delta(W_1, W_2) \leq \delta(W_1, W_3) \Rightarrow f(W_2, \mathbb{D}(W_1)) \leq f(W_3, \mathbb{D}(W_1)) \tag{8}$$

(b) δ should account for intra-query similarities; that is, if $r_i^1 > r_i^2$ and $r_j^1 < r_j^2$, the distance $\delta(W_1, W_2)$ should become smaller based on the similarity of the queries q_i and q_j, assuming the same frequencies for the other queries.

(c) δ should be symmetric; that is, $\delta(W_1, W_2) = \delta(W_2, W_1)$ for any W_1 and W_2. (This is needed for the theoretical guarantees of the BNT framework.)

[9] Remember that existing designers only take a single workload as their input parameter.

Algorithm 2: The `CliffGuard` algorithm.

Inputs: Γ: the desired degree of robustness,
δ: a distance metric defined over pairs of workloads,
W_0: initial workload,
$\mathbb{D}$: an existing (nominal) designer,
f: the cost function (or its estimate),
Output: D^*: a robust design, i.e., $D^* = \underset{D}{ArgMin} \underset{\delta(W-W_0)\leq\Gamma}{Max} f(W, D)$

$D \leftarrow \mathbb{D}(W_0)$ **// Invoke the existing designer to find a nominal design for** W_0
$P \leftarrow \{W_i \mid 1 \leq i \leq n,\ \delta(W_i, W) \leq \Gamma\}$ **// Sample some perturbed workloads in the** Γ**-neighbor of** W_0
Pick some $\alpha > 0$ **// some initial size for the descending steps**
while *true* **do**
  // Neighborhood Exploration:
  $U \leftarrow \{\tilde{W}_1, \cdots, \tilde{W}_m\}$ where $\tilde{W}_i \in P$ and $f(\tilde{W}_i, D) = \underset{W\in P}{Max} f(W,D)$ **// Pick perturbed workloads with the worst performance on** D
  // Robust Local Move:
  $W_{moved} \leftarrow$ MoveWorkload$(W_0, \{\tilde{W}_1, \cdots, \tilde{W}_m\}, f, D, \alpha)$ **//Build a new workload by moving closer to** W_0**'s worst-neighbors (see Alg. 3)**
  $D' \leftarrow \mathbb{D}(W_{moved})$ **// consider the nominal design for** W_{moved} **as an alternative design**
  if $\underset{W\in P}{Max} f(W, D') < \underset{W\in P}{Max} f(W, D)$ ***// Does*** D' ***improve on the existing design in terms of the worst-case performance?***
  then
    $D \leftarrow D'$ **// Take** D' **as your new design**
    $\alpha \leftarrow \alpha * \lambda_{success}$ (for some $\lambda_{success} > 1$) **// increase the step size for the next move along the descent direction**
  else
    $\alpha \leftarrow \alpha * \lambda_{failure}$ (for some $\lambda_{failure} < 1$) **// consider a smaller step next time**
  end
  if your time budget is exhausted or many iterations have gone with no improvements
  then
    $D^* \leftarrow D$ **// the current design is robust**
    return D^*
  end
end

(d) δ must satisfy the *triangular property*; that is, $\delta(W_1, W_2) \leq \delta(W_1, W_3) + \delta(W_3, W_2)$ for any W_1, W_2, W_3. (This is an implicit assumption in almost all gradient-based optimization techniques, including BNT.)

Before introducing a distance metric fulfilling these criteria, we need to introduce some notations. Let us represent each query as the union of all the columns that appear in it (e.g., unioning all the columns in the `select`, `where`, `group by`, and `order by` clauses). With this over-simplification, two queries will be considered identical as long as they reference the same set of columns, even if their SQL expressions, query plans, or latencies are substantially different. Using this representation, there will be only $2^n - 1$ possible queries where n is the total number of columns in the database (including all the tables). (Here, we ignore queries that do not reference any columns.) Thus, we can represent a workload W with a $(2^n - 1)$-dimensional vector $V_W = \langle r_1, \cdots, r_{2^n-1} \rangle$ where r_i represents the normalized frequency of queries that are represented by the i'th subset of the columns for $i = 1, \cdots, 2^n - 1$. With this notation, we can now introduce our Euclidean distance for database workloads as:

$$\delta_{euclidean}(W_1, W_2) = |V_{W_1} - V_{W_2}| \times S \times |V_{W_1} - V_{W_2}|^T \quad (9)$$

Here, S is a $(2^n - 1) \times (2^n - 1)$ similarity matrix, and thus $\delta_{euclidean}$ is always a real-valued number (i.e., 1×1 matrix). Each $S_{i,j}$ entry is defined as the total number of columns that are present only in q_i or q_j (but not in both), divided by $2 \cdot n$. In other words, $S_{i,j}$ is the Hamming distance between the binary representations of i and j, divided by $2 \cdot n$. Hamming distances are divided by $2 \cdot n$ to ensure a normalized distance, i.e., $0 \leq \delta_{euclidean}(W_1, W_2) \leq 1$.

One can easily verify that $\delta_{euclidean}$ satisfies criteria (b), (c), and (d). In Section 6.3, we empirically show that this distance metric also satisfies criterion (a) quite well. Finally, even though V_W is exponential in the number of columns n, it is merely a conceptual model; since V_W is an extremely sparse matrix, most of the computation in (9) can be avoided. In fact, $\delta_{euclidean}$ can be computed in $O(T^2 \cdot n)$ time and memory complexity, where T is the number of input queries (e.g., in a given query log).

Limitations — $\delta_{euclidean}$ has a few limitations. First, it does not factor in the clause in which a column appears. For instance, for fast filtering, it is more important for a materialized view to cover a column appearing in the `where` clause than one appearing only in the *select* clause. This limitation, however, can be easily resolved by representing each query as a 4-tuple $\langle v_1, v_2, v_3, v_4 \rangle$ where v_1 is the set of columns in the `select` clause and so on. We refer to this distance as $\delta_{separate}$, as we keep columns appearing in different clauses separate. $\delta_{separate}$ differs from $\delta_{euclidean}$ only in that it creates 4-tuple vectors, but it is still computed using Equation (9).

The second (and more important) limitation is that $\delta_{euclidean}$ may ignore important aspects of the SQL expression if they do not change the column sets. For example, presence of a `join` operator or using a different query plan can heavily impact the execution time, but are not captured by $\delta_{euclidean}$. In fact, as a stricter version of requirement (8), a better distance metric will be one that for all

workloads W_1, W_2, W_3 and *arbitrary* design D satisfies:

$$\begin{aligned} \delta(W_1, W_2) \leq \delta(W_1, W_3) \quad &\Rightarrow \\ |f(W_2, D) - f(W_1, D)| \quad &\leq \quad |f(W_3, D) - f(W_1, D)| \end{aligned} \tag{10}$$

In other words, the distance functions should directly match the performance characteristics of the workloads (the lower their distance, the more similar their performance). In Appendix D, we introduce a more advanced metric that aims to satisfy (11). However, in our experiments, we still use $\delta_{euclidean}$ for three reasons.

First, requirement (11) is unnecessary for our purposes. `CliffGuard` only relies on this distance metric during the neighborhood exploration and feeds *actual* SQL queries (and not just their column sets) into the existing designer. Internally, the existing designer compares the actual latency of different SQL queries, accounting for their different plans, joins, and all other details of every input query. For example, the designer ignores the less expensive queries to spend its budget on the more expensive ones.

Second, we must be able to efficiently sample the Γ-neighborhood of a given workload (see Algorithm 2, Line 2), which we can do when our cost function is $\delta_{euclidean}$. The sampling algorithm (which can be found in Appendix C) becomes computationally prohibitive when our distance metric involves computing the latency of different queries. In Section 6, we thoroughly evaluate our `CliffGuard` algorithm overall, and our distance function in particular.

The third, and final, reason is that the sole goal of our distance metric is to provide users a means to express and receive their desired degree of robustness. We show that despite its simplistic nature, $\delta_{euclidean}$ is still quite effective in satisfying (8) (see Section 6.3), and most importantly in enabling `CliffGuard` to achieve decisive superiority over existing designers (see Section 6.4).

In the end, we note that *quantifying* the amount of change in SQL workloads is a research direction that will likely find many other applications beyond robust physical designs, e.g., in workload monitoring [49, 76], auto-tuning [38], or simply studying database usage patterns. We believe that $\delta_{euclidean}$ is merely a starting point in the development of more advanced and application-specific distance metrics for database workloads.

Algorithm 3: The subroutine for moving a workload.

Inputs: W_0: an initial workload,
$\{\tilde{W}_1, \cdots, \tilde{W}_m\}$: workloads to merge with W_0,
f: the cost function (or its estimate),
D: a given design,
α: a scaling factor for the weight ($\alpha > 0$)

Output: W_{moved}: a new (merged) workload which is closer to $\{\tilde{W}_1, \cdots, \tilde{W}_N\}$ than W_0, i.e.,
$\Sigma_i \delta(\tilde{W}_i, W_{moved}) < \Sigma_i \delta(\tilde{W}_i, W_0)$

Subroutine `MoveWorkload` $(W_0, \{\tilde{W}_1, \cdots, \tilde{W}_m\}, f, D, \alpha)$

2 $W_{moved} \leftarrow \{\}$
3 $Q \leftarrow$ the set of all queries in W_0 and $\tilde{W}_1, \cdots, \tilde{W}_m$ workloads
4 **foreach** query $q \in Q$ **do**
5 $f_q \leftarrow f(\{q\}, D)$ **// the cost of query q using design D**
6 $\omega_q \leftarrow (f_q \cdot \sum_{i=1}^{m} \text{weight}(q, \tilde{W}_i))^{\alpha} + \text{weight}(q, W_0)$
7 $W_{moved} \leftarrow W_{moved} \cup \{(q, \omega_q)\}$
end
9 **return** W_{moved}
end

6. EXPERIMENTAL RESULTS

The purpose of our experiments in this section is to demonstrate that (i) real world workloads can vary over time and be subject to a great deal of uncertainty (Section 6.2), (ii) despite its simplicity, our distance metric $\delta_{euclidean}$ can reasonably capture the performance implications of a changing workload (Section 6.3), and most importantly (iii) our robust design formulation and algorithm improve the performance of the state-of-the-art industrial designers by up to an order of magnitude, without having to modify the internal implementations of these commercial tools (Section 6.4). We also study different degrees of robustness (Section 6.5). Additional experiments are deferred to Appendix A, where we evaluate the effects of different distance functions and other parameters on `CliffGuard`'s performance, and show that `CliffGuard`'s processing overhead is negligible compared to that of the deployment phase.

6.1 Experimental Setup

We have implemented `CliffGuard` in Java. We tested our algorithm against Vertica's database designer (called DBD [75, 83]) and DBMS-X's designer as two of the most heavily-used state-of-the-art commercial designers, as well as two other baseline algorithms (introduced later in this section). For Vertica experiments, we used its community edition and invoked its DBD and query optimizer via a JDBC driver. Similarly, we used DBMS-X's latest API. We ran each experiment on two machines: a server and a client. The server ran a copy of the database and was used for testing different designs. The client was used for invoking the designer and sending queries to the server. We ran the Vertica experiments on two Dell machines running Red Hat Enterprise Linux 6.5, each with two quad-core Intel Xeon 2.10GHz processors. One of the machines had 128GB memory and 8×4TB 7.2K RPM disks (used as server) and the other had 64GB memory and 4×4TB 7.2K RPM disks. For DBMS-X experiments, we used two Azure Standard Tier A3 instances, each with a quad-core AMD Opteron 4171 HE 2.10GHz processor, 7GB memory, and 126GB virtual disks. In this section, when not specified, we refer to our Vertica experiments.

Workloads[10] — We conducted our experiments on a real-world (R1) workload and two synthetic ones (S1 and S2). R1 belongs to one of the largest customers of the Vertica database, composed of 310 tables and $430{+}K$ time-stamped queries issued between March 2011 and April 2012 out of which $15.5K$ queries conform to their latest schema (i.e., can be parsed). We did not have access to their original dataset but we did have access to their data distribution, which we used to generate a 151GB dataset for our Vertica experiments. Since we did not have access to any real workloads from DBMS-X's customers, we used the same query log but on a smaller dataset (20GB) given the smaller memory capacity of our Azure instances (compared to our Dell servers). We also created two synthetic workloads, called S1 and S2, as follows. We used the same schema and dataset as R1, but chose different subsets and relative ordering of R1 queries to artificially cause different degrees of workload change. Table 1 reports basic statistics on the amount workload changes (in terms of $\delta_{euclidean}$) between consecutive windows of queries where each window was 28 days (different window sizes are studied in Section 6.2). S1 queries were chosen to mimic a workload with minimal change over time (between $0.1m$ and m, where m is the minimum change observed in R1). S2 queries were chosen to exhibit the same range of $\delta_{euclidean}$ as R1 but more uniformly. More detailed analysis of these workloads will be presented in the subsequent sections.

[10]Common benchmarks (e.g., TPC-H) are not applicable here as they only contain a few queries, and do not change over time.

Workload	Min $\delta(W_i,W_{i+1})$	Max $\delta(W_i,W_{i+1})$	Avg $\delta(W_i,W_{i+1})$	Std $\delta(W_i,W_{i+1})$
R1	m=0.00016	M=0.00311	0.00120	0.00122
S1	0.1m	m	0.00006	0.00003
S2	m	M	0.00178	0.00063

Table 1: Summary of our real-world and synthetic workloads.

Algorithms Compared — We divided the queries according to their timestamps into 4-week windows, $W_0, W_1, \cdots$. We re-designed the database at the end of each month to simulate a tuning frequency of a month (a common practice, based on our oral conversations). In other words, we fed W_i queries into each of the following designers and used the produced design to process W_{i+1} (except for `FutureKnowingDesigner`; see below).

1. `NoDesign`: A dummy designer that returns an empty design (i.e., no projections). Using `NoDesign` all queries simply scan the default super-projections (which contain all the columns), providing an upper limit on each query's latency.

2. `ExistingDesigner`: The nominal designer shipped with commercial databases. For instance, Vertica's DBD [83] recommends a set of projections while DBMS-X's designer finds various types of indices and materialized views. We used these state-of-the-art designers as our main baselines.

3. `FutureKnowingDesigner`: The same designer as `ExistingDesigner`, except that instead of feeding queries from W_i and testing on W_{i+1}, we both feed and test it on W_{i+1}. This designer signifies the best performance achievable where the designer knows exactly which queries to expect in the future and optimize for.

4. `MajorityVoteDesigner`: A designer that uses sensitivity analysis to identify elements of the nominal design that are *brittle* against changes of workload. This designer uses the same technique as `CliffGuard` to explore the local neighborhood of the current W_i, and generate a set of perturbed workloads $W_i^1, \cdots, W_i^n$. Then, it invokes the `ExistingDesigner` to suggest an optimal design for each W_i^j. Finally, for each structure (e.g., index, materialized view, projection) s, `MajorityVoteDesigner` counts the number of times that s has appeared in the nominal design of the neighbors, and selects those structures that have appeared in different designs most frequently. The idea behind this heuristic is that structures that appear in the optimal design of fewer neighbors (have fewer votes) are less likely to remain beneficial when the future workload changes.

5. `OptimalLocalSearchDesigner`: Similar to `MajorityVoteDesigner`, this designer starts by searching the local neighborhood of the given workload and generating perturbed workloads. However, instead of selecting structures that have been voted for by the most number of neighbors, this designer takes the union of the queries in the neighboring workloads as the *expectation* (i.e., representative) of the future workload, say $\bar{W}$. This algorithm then solves an Integer Linear Program to find an optimal set of structures that fit in the budget and minimize the cost of $\bar{W}$.[11]

7. `CliffGuard`: Our robust database designer from Section 4.

Note that DBD and DBMS-X's designer (`ExistingDesigner`) are our goal standards as the state-of-the-art designers currently used in the industry. However, we also aim to answer the following question. How much of `CliffGuard`'s overall improvement over nominal designers is due to its exploration of the initial workload's local neighborhood, and how much is due to its carefully selected descent direction and step sizes in moving away from the worst neighbors? Since `MajorityVoteDesigner` and `OptimalLocalSearchDesigner` use the same neighborhood sampling strategy as `CliffGuard` but employ greedy and local search heuristics, we will be able to break down the contribution of `CliffGuard`'s various components to its overall performance.

[11] A greedy version of this algorithm and a detailed description of the other baselines can be found in Appendix F.

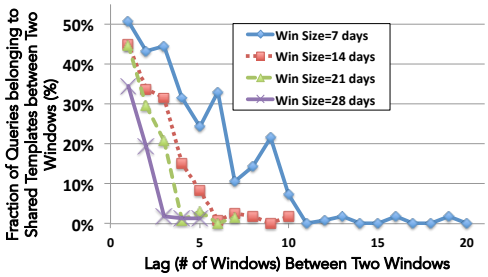


Figure 5: Many workloads drift over time ($15.5K$ queries, 6 months).

Since Vertica automatically decides on the storage budget (50GB in our case), we used the same budget for the other algorithms too. For DBMS-X experiments, we used a maximum budget of 10GB (since the dataset was smaller). Also, unless otherwise specified, we used n=20 samples in all algorithms involving sampling, and 5 iterations, $\lambda_{success} = 5$, and $\lambda_{success} = 0.5$ in `CliffGuard`.

6.2 Workloads Change Over Time

First, we studied if and how much our real workload has changed over time. While OLTP and *reporting* queries tend to be more repetitive (often instantiated from a few templates with different parameters), analytical and exploratory workloads tend to be less predictable (e.g., Hive queries at Facebook are reported to access over 200–450 different subsets of columns [6]). Likewise, in our analytical workload R1, we observed that queries issued by users have constantly drifted over time, perhaps due to the changing nature of their company's business needs.

Figure 6.1 shows the percentage of queries that belonged to templates that were shared among each pair of windows as the time lag grew between the two windows. Here, we have defined templates by stripping away the query details except for the sets of columns used in the `select`, `where`, `group by`, and `order by` clauses. This is an overly optimistic analysis assuming that queries with the same column sets in their respective clauses will exhibit a similar performance. However, even with this optimistic assumption, we observed that for a window size of one week, on average only 51% of the queries had a similar counterpart between consecutive weeks. This percentage was only 35% when our window was 4 weeks. Regardless of the window size, this commonality drops quickly as the time lag increases, e.g., after 2.5 months less than 10% of the queries had similar templates appearing in the past. The unpredictability of analytical workloads underlines the important role of a robust designer. We show in Section 6.4 that failing to take into account this potential change (i.e., uncertainty) in our target workload has a severe impact on the performance of existing physical designers — one that we aim to overcome via our robust designs.

6.3 Our Distance Metric Is Sound

In Section 5, we introduced our distance metric $\delta_{euclidean}$ to concisely quantify the dissimilarity of two SQL workloads. While we do not claim that $\delta_{euclidean}$ is an ideal one (see Section 5), here we show that it is sound. That is, in general:

$$\delta(W_0, W) \leq \delta(W_0, W') \Rightarrow f(W, \mathbb{D}(W_0)) \leq f(W', \mathbb{D}(W_0))$$

which means that a design made for W_0 is more suitable for W than it is for W', i.e., W will experience a lower latency than W'.

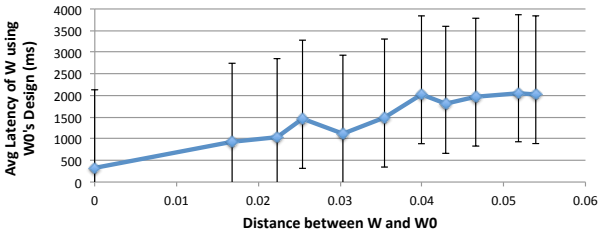


Figure 6: Performance decay of a window W on a design made for another window W_0 is highly correlated with their distance.

Figure 6 reports an experiment where we chose 10 different starting windows as our W_0 and created a number of windows with different distances from W_0. The curve (error bar) shows the average (range) of the latencies of these different windows for each distance. This plot indicates a strong correlation and monotonic relationship between performance decay and $\delta_{euclidean}$. Later, in Section 6.4, we show that even with this simplistic distance metric, our `CliffGuard` algorithm can consistently improve on Vertica's latest designer by severalfold.

6.4 Quality of Robust vs. Nominal Designs

In this section, we turn to the most important questions of this paper: is our robust designer superior to state-of-the-art designers? And, if so, by what measure? We compared these designers using all 3 workloads. In R1, out of the 15.5K queries, only 515 could benefit from a physical design, i.e., the remaining queries were either trivial (e.g., `select version()`) or returned an entire table (e.g., '`select * from T`' queries with no filtering used for backup purposes) in which case they always took the same time as they only used the super-projections in Vertica and table-scans in DBMS-X. Thus, we only considered queries for which there existed an *ideal* design (no matter how expensive) that could improve on their bare table-scan latency by at least a factor of 3×.

Figure 7 summarizes the results of our performance comparison on Vertica, showing the average and maximum latencies (both averaged over all windows) for all three workloads. On average, `MajorityVoteDesigner` improved on the existing designer by 13%, while `OptimalLocalSearchDesigner`'s performance was slightly worse than Vertica's DBD. However, `CliffGuard` was superior to the existing designer by an astonishing margin: on average, it cut down the maximum latency of each window by 39.7× and 13.7× for R1 and S2, respectively. Interestingly, for these workloads, even `CliffGuard`'s average-case performance was 14.3× and 5.3× faster than `ExistingDesigner`. The last result is surprising because our `CliffGuard` is designed to protect against worst-case scenarios and ensure a predictable performance. However, improvement even on the average case indicates that the design space of a database is highly non-convex — and as such can easily delude a designer into a local optimum. Thus, by avoiding the proximity of bad neighbors, `CliffGuard` seems to find designs that are also more globally optimal. In fact, for S2, Figure 7(c) shows that `CliffGuard` is only 30% worse than a hypothetical, ideal world where future queries are precisely known in advance (i.e., the `FutureKnowingDesigner`). For S1, however, `CliffGuard`'s improvement over `ExistingDesigner` is more modest: 1.5× improvement for worst-case latency and 1.2× for average latency. This is completely expected since S1 is designed to exhibit no or little change between different windows (refer to Table 1). This is the ideal case for a nominal designer since the amount of uncertainty across workloads is so negligible that even our hypothetical `FutureKnowingDesigner` cannot improve much on the nominal designer. Thus, averaging over all three workloads, compared to `ExistingDesigner`, `CliffGuard` improves the average and worst-case latencies by 6.9× and 18.3×, respectively.

Figure 10 reports a similar experiment for workload R1 but for DBMS-X. (DBMS-X experiments on workloads S1 and S2 can be found in Appendix A.3.) Even though DBMS-X's designer has been fine-tuned and optimized over the years, `CliffGuard` still improves its worst-case and average-case performances by 2.5–5.2× and 2–3.2×, respectively. This is quite encouraging given that `CliffGuard` is still in its infancy stage of development and treats the database as a black-box. While still significant, the improvements here are smaller than those observed with Vertica. This is due to several heuristics used in DBMS-X's designer (such as omitting workload details) that prevent it from overfitting its input workload. However, this also shows that dealing with such uncertainties in a principled framework can be much more effective.

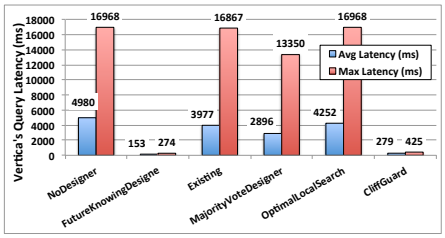
These experiments confirm our hypothesis that failing to account for workload uncertainty can have significant consequences. For example, for R1 on Vertica, `ExistingDesigner` is on average only 25% better than `NoDesign` (with no advantage for the worst-case). Note that here the database was re-designed every month, which means even this slight advantage of `ExistingDesigner` over `NoDesign` would quickly fade away if the database were to be re-designed less frequently (as the distance between windows often increases with time; see Figure 6.1). These experiments show the ample importance of re-thinking and re-architecting the existing designers currently shipped and used in our database systems.

6.5 Effect of Robustness Knob on Performance

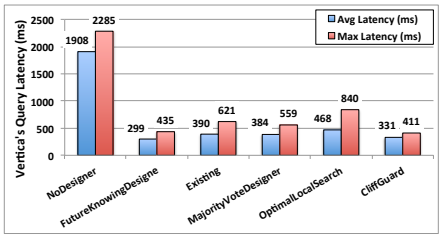
To study the effect of different levels of robustness, we varied the Γ parameter in our algorithm and measured the average and worst-case performances in each case. The results of this experiment for workloads R1 and S2 are shown in Figures 8 and 9, respectively. (As reported in Section 6.4, workload S1 contains minimal uncertainty and thus is ruled out from this experiment, i.e., the performance difference between `ExistingDesigner` and `CliffGuard` remains small for S1). Here, experiments on both workloads confirm that requesting a large level of robustness will force `CliffGuard` to be overly conservative, eliminating its margin of improvement over `ExistingDesigner`. Note that in either case `CliffGuard` still performs no worse than `ExistingDesigner`, which is due to two reasons. First, `ExistingDesigner` is only marginally better than `NoDesign` (refer to Section 6.4) and as Γ increases, its relevance for the actual workload (which has a much lower $\delta_{euclidean}$) degrades. As a result, both designers approach `NoDesign`'s performance, which serves an upper bound on latency (i.e., unlike theory, latencies are always bounded in practice, due to the finite cost of the worst query plan). The second reason is that, during each iteration of `CliffGuard` (unlike BNT), our new workload always contains the original workload which ensures that even when Γ is large, the designer will not completely ignore the original workload (see Algorithm 3). Also, as expected, as Γ approaches zero, `CliffGuard`'s performance again approaches that of a nominal designer.

7. RELATED WORK

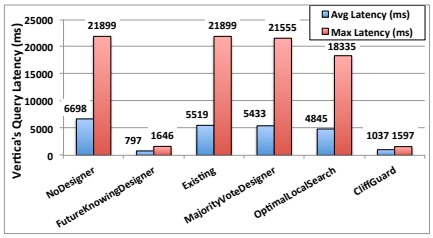
There has been much research on physical database design problems, such as the automatic selection of materialized views [10, 14, 18, 43, 61, 67, 77, 84, 87], indices [33, 34, 47, 64, 72, 81], or both [9, 13, 38, 56, 89]. Also, most modern databases come with designer tools, e.g., Tuning Wizard in Microsoft SQL Server [10], IBM DB2's Design Advisor [89], Teradata's Index Wizard [26], and Oracle's SQL Tuning Adviser [38]. Other types of design problems include project selection in columnar databases [40, 57, 83] and stratified sample selection in approximate databases [4, 6, 17, 29]. All these designers are nominal and assume that their target workload is precisely known. Since future queries are often not



(a) Real-world workload R1 on Vertica.



(b) Synthetic static workload S1 on Vertica.



(c) Synthetic drifting workload S2 on Vertica.

Figure 7: Average and worst-case performances of designers for Vertica, averaged over all windows, for workloads R1, S1, and S2.

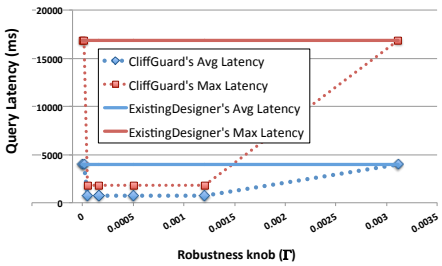


Figure 8: Different degrees of robustness for Workload R1.

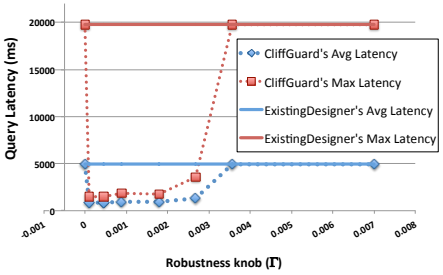


Figure 9: Different degrees of robustness for Workload S2.

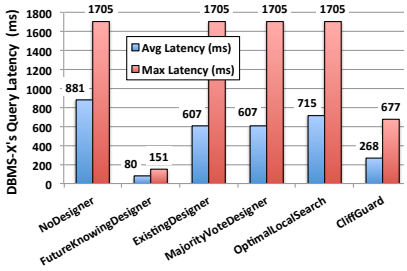


Figure 10: Performance of different designers for DBMS-X on workload R1.

known in advance, these tools optimize for past queries as approximations of future ones. By failing to take into account the fact that a portion of those queries will be different in the future, they produce designs that are sub-optimal and brittle in practice. To mitigate some of these problems, a few heuristics [35] have been proposed to compress and summarize the workload [30, 55] or modify the query optimizer to produce richer statistics [44]. However, these approaches are not principled and thus, do not necessarily guarantee robustness. In contract, `CliffGuard` takes the possible changes of workload into account in a principled manner, and directly maximizes the robustness of the physical design.

To avoid these limitations, adaptive indexing schemes [45, 46, 48, 53, 76]) take the other extreme by avoiding the offline physical design, and instead, creating and adjusting indices incrementally, *on demand*. Despite their many merits, these schemes do not have a mechanism to incorporate prior knowledge under a bounded amount of uncertainty. Also, one still needs to decide which subsets of columns to build an adaptive index on. For these reasons, most commercial databases still rely on their offline designers. In contrast, `CliffGuard` uses RO theory to directly minimize the effect of uncertainty on optimality, and guarantee robustness.

The effect of uncertainty (caused by cost and cardinality estimates) has also been studied in the context of query optimization [16, 31, 37, 42, 66, 74] and choosing query plans with a bounded worst-case [15]. None of these studies have addressed uncertainties caused by workload changes, or their impact on physical designs. Also, while these approaches produce plans that are more predictable, they are not principled in that they do not directly maximize robustness, i.e., they do not guarantee robustness even in the context of query optimization. Finally, most of these heuristics are specific to a particular problem and do not generalize to others.

Theory of robust optimization has taken many strides in recent years [21, 22, 23, 36, 41, 58, 88] and has been applied to many other disciplines, e.g., supply chain management [21], circuit [73] and antenna [63] design, power control [51], control theory [20], thin-film manufacturing [24], and microchip architecture [71]. However, to the best of our knowledge, this paper is the first application of RO theory in a database context (see Section 4.2).

8. CONCLUSION AND FUTURE WORK

The state-of-the-art database designers rely on heuristics that do not account for uncertainty, and hence produce sub-optimal and brittle designs. On the other hand, the principled framework of robust optimization theory, which has witnessed remarkable advances over the past few years, has largely dealt with problems that are quite different in nature than those faced in databases. In this paper, we presented `CliffGuard` to exploit these techniques in the context of physical design problems in a columnar database. We compared our algorithm to a state-of-the-art commercial designer using several real-world and synthetic workloads. In summary, compared to Vertica's state-of-the-art designer, our robust designer reduces the average and maximum latency of queries by up to $5\times$ and $11\times$, respectively. Similarly, `CliffGuard` improves upon DBMS-X's designer by 3–$5\times$. Since `CliffGuard` treats the existing designer as a block-box, with no modifications to the database internals, an interesting future direction is to extend `CliffGuard` to other major DBMSs with other types of design problems.

Acknowledgements

This work is in part supported by Amazon AWS and Microsoft Azure. The authors would like to thank the anonymous reviewers for their insightful feedback, Michael Stonebraker and Samuel Madden for their early contributions, Stephen Tu for his SQL parser, Shiyong Hu and Ruizhi Deng for their help with implementing our distance metrics, and Yingying Zhu for plotting Figure 4. The authors are also grateful to Andrew Lamb and Vivek Bharathan (for helping with Vertica experiments), and Alice Tsay and Suchee Shah (for their comments on this manuscript).

9. REFERENCES

[1] CliffGuard: A General Framework for Robust and Efficient Database Optimization. `http://www.cliffguard.org`.

[2] GNU Linear Programming Kit. `www.gnu.org/s/glpk/`.

[3] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *VLDB*, September 1999.

[4] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *ACM SIGMOD*, May 2000.

[5] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[7] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and it's done: Interactive queries on very large data. *PVLDB*, 2012.

[8] S. Agrawal, N. Bruno, S. Chaudhuri, and V. R. Narasayya. Autoadmin: Self-tuning database systems technology. *IEEE Data Eng. Bull.*, 29(3), 2006.

[9] S. Agrawal and et. al. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[10] S. Agrawal and et. al. Materialized view and index selection tool for microsoft sql server 2000. In *SIGMOD*, 2001.

[11] I. Alagiannis, D. Dash, K. Schnaitter, A. Ailamaki, and N. Polyzotis. An automated, yet interactive and portable db designer. In *SIGMOD*, 2010.

[12] I. Alagiannis, S. Idreos, and A. Ailamaki. H2o: a hands-free adaptive store. In *SIGMOD*, 2014.

[13] K. Aouiche and J. Darmont. Data mining-based materialized view and index selection in data warehouses. *Journal of Intelligent Information Systems*, 33(1), 2009.

[14] K. Aouiche, P.-E. Jouve, and J. Darmont. Clustering-based materialized view selection in data warehouses. In *Advances in Databases and Information Systems*, 2006.

[15] M. Armbrust and et. al. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5, 2011.

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

[17] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *VLDB*, 2003.

[18] E. Baralis and et. al. Materialized view selection in a multidimensional database. In *VLDB*, volume 97, 1997.

[19] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust optimization*. Princeton University Press, 2009.

[20] D. Bertsimas and D. B. Brown. Constrained stochastic lqc: a tractable approach. *Automatic Control, IEEE Transactions on*, 52(10), 2007.

[21] D. Bertsimas and et. al. Theory and applications of robust optimization. *SIAM*, 53, 2011.

[22] D. Bertsimas, O. Nohadani, and K. M. Teo. Robust nonconvex optimization for simulation-based problems. *Operations Research*, 2007.

[23] D. Bertsimas and M. Sim. The price of robustness. *Operations research*, 52(1), 2004.

[24] J. R. Birge and et. al. Improving thin-film manufacturing yield with robust optimization. *Applied optics*, 50, 2011.

[25] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[26] D. P. Brown, J. Chaware, and M. Koppuravuri. Index selection in a database system, Mar. 3 2009. US Patent 7,499,907.

[27] N. Bruno, S. Chaudhuri, A. C. König, V. R. Narasayya, R. Ramamurthy, and M. Syamala. Autoadmin project at microsoft research: Lessons learned. *IEEE Data Eng. Bull.*, 34(4), 2011.

[28] C. Chatfield. *Time-series forecasting*. CRC Press, 2002.

[29] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.

[30] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *SIGMOD*, 2002.

[31] S. Chaudhuri, H. Lee, and V. R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, 2010.

[32] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[33] S. Chaudhuri, V. Narasayya, M. Datar, et al. Linear programming approach to assigning benefit to database physical design structures, Nov. 21 2006. US Patent 7,139,778.

[34] S. Chaudhuri and V. R. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In *VLDB*, volume 97, 1997.

[35] A. N. K. Chen and et. al. Heuristics for selecting robust database structures with dynamic query patterns. *EJOR*, 168, 2006.

[36] X. Chen, M. Sim, and P. Sun. A robust optimization perspective on stochastic programming. *Operations Research*, 55, 2007.

[37] F. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *PODS*, 1999.

[38] B. e. Dageville. Automatic sql tuning in oracle 10g. In *VLDB*, 2004.

[39] K. Deb. Geneas: A robust optimal design technique for mechanical component design. 1997.

[40] X. Ding and J. Le. Adaptive projection in column-stores. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 4, 2011.

[41] I. Doltsinis and et. al. Robust design of non-linear structures using optimization methods. *Computer methods in applied mechanics and engineering*, 194, 2005.

[42] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

[43] C. A. Galindo-Legaria and M. M. Joshi. Cost based materialized view selection for query optimization, Jan. 21 2003. US Patent 6,510,422.

[44] K. E. Gebaly and A. Aboulnaga. Robustness in automatic

physical database design. In *Advances in database technology*, 2008.

[45] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *ICDEW*, 2010.

[46] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *ICEDT*, 2010.

[47] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *Data Engineering, 1997. Proceedings. 13th International Conference on*. IEEE, 1997.

[48] F. Halim and et. al. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5, 2012.

[49] M. Holze, A. Haschimi, and N. Ritter. Towards workload-aware self-management: Predicting significant workload shifts. In *CDEW*, 2010.

[50] M. Holze and N. Ritter. Autonomic databases: Detection of workload shifts with n-gram-models. In *Advances in Databases and Information Systems*, 2008.

[51] K.-L. Hsiung, S.-J. Kim, and S. Boyd. Power control in lognormal fading wireless channels with uptime probability specifications via robust geometric programming. In *American Control Conference*, 2005.

[52] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[53] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[54] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003.

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

[56] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. Stallmann. View and index selection for query-performance improvement: quality-centered algorithms and heuristics. In *CIKM*, 2008.

[57] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12), 2012.

[58] K.-H. Lee and G.-J. Park. A global robust optimization using kriging based approximation model. *JSME*, 49, 2006.

[59] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *SIGMOD*, 2014.

[60] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *PVLDB*, 5(11), 2012.

[61] W. Liang, H. Wang, and M. E. Orlowska. Materialized view selection under the maintenance time constraint. *Data & Knowledge Engineering*, 37(2), 2001.

[62] S. S. Lightstone, G. Lohman, and D. Zilio. Toward autonomic computing with db2 universal database. *SIGMOD Record*, 31(3), 2002.

[63] R. G. Lorenz and S. P. Boyd. Robust minimum variance beamforming. *Signal Processing, IEEE Transactions on*, 53(5), 2005.

[64] C. Maier and et. al. Parinda: an interactive physical designer for postgresql. In *ICEDT*, 2010.

[65] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD*, 41(1), 2012.

[66] V. Markl and et. al. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[67] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Record*, 30(2), 2001.

[68] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, 2013.

[69] B. Mozafari, C. Curino, and S. Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[70] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[71] G. Palermo, C. Silvano, and V. Zaccaria. Robust optimization of soc architectures: A multi-scenario approach. In *Embedded Systems for Real-Time Multimedia*, 2008.

[72] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDE Workshop*, 2007.

[73] D. Patil, S. Yun, S.-J. Kim, A. Cheung, M. Horowitz, and S. Boyd. A new method for design of robust digital circuits. In *ISQED*, 2005.

[74] V. Raman and et. al. Constant-time query processing. In *ICDE*, 2008.

[75] A. RASIN and et. al. Automatic vertical-database design, 2008. WO Patent.

[76] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE Workshop*, 2007.

[77] A. Shukla, P. Deshpande, J. F. Naughton, et al. Materialized view selection for multidimensional datasets. In *VLDB*, volume 98, 1998.

[78] M. Stonebraker and et. al. C-store: a column-oriented dbms. In *VLDB*, 2005.

[79] Z. A. Talebi, R. Chirkova, and Y. Fathi. An integer programming approach for the view and index selection problem. *Data & Knowledge Engineering*, 83, 2013.

[80] J. Tu, K. K. Choi, and Y. H. Park. A new study on reliability-based design optimization. *Journal of Mechanical Design*, 121(4), 1999.

[81] G. Valentin and et. al. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[82] A. van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

[83] R. Varadarajan, V. Bharathan, A. Cary, J. Dave, and S. Bodagala. Dbdesigner: A customizable physical design tool for vertica analytic database. In *ICDE*, 2014.

[84] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 33(4), 2003.

[85] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. Abs: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.

[86] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.

[87] C. Zhang and J. Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *DataWarehousing and Knowledge Discovery*. 1999.

[88] Y. Zhang. General robust-optimization formulation for nonlinear programming. *JOTA*, 132, 2007.

[89] D. C. Zilio and et. al. Recommending materialized views and

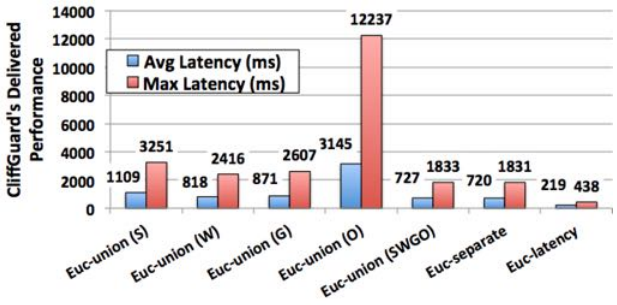


Figure 11: The effect of different distance functions on `CliffGuard`'s effectiveness (on workload R1).

indexes with the ibm db2 design advisor. In *ICAC*, 2004.

APPENDIX

The goal of this section is to provide the interested reader with additional details on various aspects of our approach. Several important experiments are presented in Appendix A. Appendix B provides the optimization formulation used in the BNT framework for finding the (steepest) descent direction [22]. (As mentioned in Section 4.2, this formulation cannot be directly used for finding robust database designs.) Appendix C provides the detailed sampling strategy used in `CliffGuard`, when the Γ-neighborhood is expressed using distance $\delta_{euclidean}$ (introduced in Section 5). Appendix D provides a few examples and a more detailed discussion on various alternatives for defining database-specific distance metric. Additional related work on physical design is overviewed in Appendix E. Finally, in Section F, we provide a more formal description of the baseline algorithms used in Section 6.

A. ADDITIONAL EXPERIMENTS

In this appendix, we study the effects of different distance metrics and other parameters on `CliffGuard`'s performance, in Sections A.1 and A.2, respectively. We report additional experiments using DBMS-X in Section A.3. Finally, the offline times for various design algorithms are presented in Section A.4.

A.1 The Distance Metric's Effect on CliffGuard

To study the implications of our choice of distance function on `CliffGuard`'s effectiveness, we have repeated our experiments on R1 using various distance metrics. The results are shown in Figure 11. Here, Euc-union is our $\delta_{euclidean}$ metric defined in Section 5, annotated by the clauses used in its binary vector. For instance, in Euc-union (S), we only consider the set of columns appearing in the `select` clause while in Euc-union (SWGO) represents our default choice where we take the union of all columns appearing in the `select`, `where`, `group by`, and `order by` clauses. Euc-separate is our extension of $\delta_{euclidean}$, called $\delta_{separate}$, which was defined in Section 5. Finally, Euc-latency is another extension of our Euclidean distance that besides the intra-column similarities, it also accounts for the actual latency of the queries to differentiate queries that share the same set of columns but differ in their latencies significantly. (The latency-aware metric is formally defined as $\delta_{latency}$ in Appendix D.)

As expected, `CliffGuard`'s reaches its best performance when using Euc-latency since it can quantify the changes of the workload more accurately. Interestingly, despite using more bits, the difference between Euc-separate and Euc-union (SWGO) is quite negligible. When constrained to only consider the columns in one of the clauses, the `where` and `group by` clauses seem to be the most informative ones to the `CliffGuard`, as they inform the underlying nominal designer which sets of columns will be filtered on or grouped. Columns in the `order by` clause are only involved in post-processing of the queries and hence, do not prove as useful. The `select` clauses, however, seem suprisingly informative. Upon closer examination, we have discovered that this is because the majority of the columns referenced in the `where` and `group by` clauses of our queries also appear in the `select` clause. Thus, the `select` clause indirectly informs the designer about the filtering and group columns. In summary, while Euc-latency appears to the be the most effective in capturing the workload uncertainty, `CliffGuard`'s default choice is still Euc-union (SWGO) since the former cannot be efficiently computed and hence, is impractical. However, we can compute $\delta_{euclidean}$ much more efficiently; see Section 5 and Appendix D).

A.2 Studying various Parameters in CliffGuard

Besides Γ, which is a user-provided parameter, there are two other important parameters in `CliffGuard`. Figure 12 studies the effect of varying the number of sampled workloads (i.e., n in Line 2 of Algorithm 2) on both average and worst-case performances, indicating that with as few as 10 sampled workloads, `CliffGuard` is able to infer a general direction of descent which will make it farther from its worst-neighbors. We also varied the number of iterations, shown in Figure 13. Surprisingly, `CliffGuard` is capable of moving away from its worst-neighbors and quickly reaching a local robust optimum within a few iterations. We believe this is due to BNT's principled framework, which by moving along the (steepest) descent direction guarantees fast convergence in practice. For this reason, the default number of iterations in `CliffGuard` is currently 5, as we rarely observe any improvement after that (i.e., we reach a robust solution earlier).

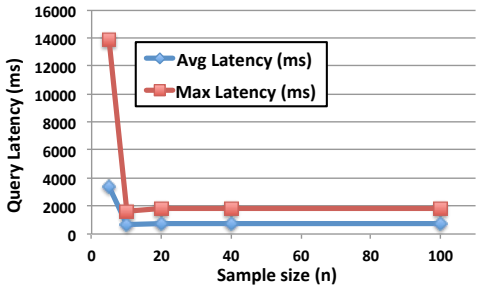


Figure 12: The effect of sample size on `CliffGuard`'s performance.

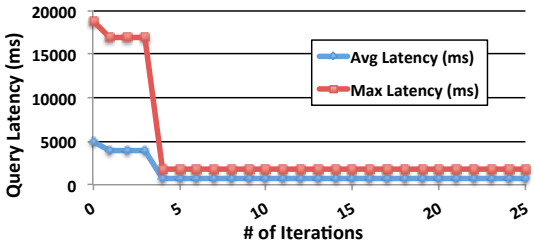


Figure 13: The effect of number of iterations on `CliffGuard`'s performance.

A.3 Additional Comparisons for DBMS-X

We have compared our various baselines for DBMS-X on all three workloads. In Section 6.4, we reported the results for the real

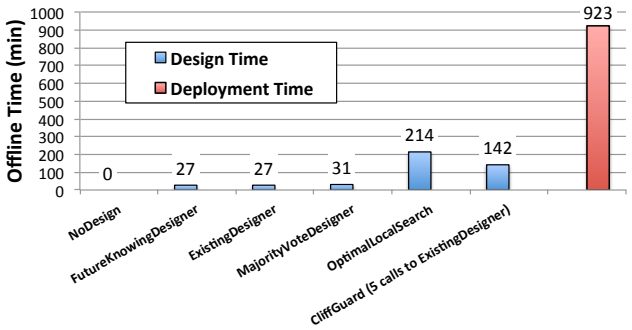


Figure 14: Offline-time taken by each designer compared to the deployment time.

workload R1. The results for workloads S1 and S2 can be found in Figure 15.

A.4 Offline Processing Time

The last question that we study is how expensive a robust designer is compared a nominal one, and also compared to the deployment phase (i.e., the actual creation of the selected projections). Figure 14 shows that our robust designer, `CliffGuard`, takes 2.3 hours to finish while `ExistingDesigner` takes about 30 minutes. This is of course expected since `CliffGuard` is an iterative algorithm that invokes `ExistingDesigner` in each iteration (refer to our design principles in Section 2). As noted earlier, the maximum number of iterations is `CliffGuard` is currently 5. The fastest baseline is `MajorityVoteDesigner`, which only performs counting after the initial design.

Note that while finding a robust design takes $5\times$ longer than finding a nominal one, this is still an attractive price to pay for robustness due to the following reasons. First, the superiority of `CliffGuard` over `ExistingDesigner` is *not* due to the longer time that the former takes, but rather the different type of design that it produces. In other words, we cannot provide `ExistingDesigner` with more time to find a better design as it has already found the *nominal* optimum after 30 minutes and will not search for a robust design since it is not a robust designer (remember than `ExistingDesigner` does not take a time budget and finishes when it has completed its search). Second, finding a physical design is an *offline* process that only occurs monthly or a few times a year. The reason why databases are not re-designed more frequently is not because of the cost of finding a new design, but rather due to the prohibitive cost of *creating and deploying* a new design. For instance, our database takes more than 15 hours to completely deploy a new design, i.e., the design time is negligible compared to the actual deployment time. (Also, as a database grows in size, design time remains the same but deployment cost grows linearly). Finally, many users will be willing to pay a $5\times$ penalty in offline processing time to win a 5–$10\times$ improvement in their online query processing latency (see Section 6.4 for our latency comparison).

B. FINDING DESCENT DIRECTIONS

Bertsimas et al. [22] use a second-order cone program (SOCP) to find the (steepest) descent direction, given all the worst-neighbors of the current decision point. Their formulation is presented in Algorithm 4. The descent direction must form the maximum angle θ_{max} with all the $(u_i) - x$ vectors. The intuition is that when such a descent direction exists, θ_{max} is always greater than 90°(refer to Figure 3(a)) because of the $\vec{d}'(u_i - x) \leq \beta \leq -\epsilon < 0$ constraint. In this algorithm, $\beta \leq 0$ is not used in place of $\beta < 0$ to exclude the trivial solution which is the zero vector $\vec{\mathbf{d}}^* = \mathbf{0}$. As noted earlier, SOCPs can be solved efficiently by most open-source and commercial solvers (via the so-called interior point methods [25]).

Algorithm 4: The subroutine for finding the descent direction.

Inputs: x: the initial point,
U: the set of all worst-neighbors of x
Output: $\vec{\mathbf{d}}^*$: a decent direction that moves away from all $\vec{\Delta}x_i$ vectors where $\vec{\Delta}x_i = (u_i - x)$ for every $u_i \in U$

Subroutine `FindDescentDirection` (x, U)

// Solve the following second-order cone program (SOCP)

$$(\vec{\mathbf{d}}^*, \beta^*) \leftarrow \underset{\vec{d},\beta}{ArgMin}\ \beta \qquad (11)$$

such that:
$$||\vec{d}||_2 \leq 1,$$
$$\vec{d}'(u_i - x) \leq \beta \quad \forall\, u_i \in U$$
$$\beta \leq -\epsilon \quad (\epsilon \text{ is a small positive scalar})$$

if problem (11) is infeasible
then
  return null **// No descent direction exists**
else
  return $\vec{\mathbf{d}}^*$
end
end

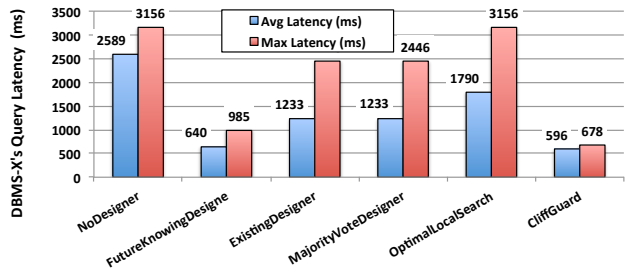
C. SAMPLING THE WORKLOAD SPACE

To implement `CliffGuard` we must be able to efficiently sample the workload space (Algorithm 2, Line 2). In other words, given a workload W_0 and a certain distance Γ we must find $n > 1$ neighbors $W_1, \cdots, W_n$ such that $\delta(W_0, W_i) \leq \Gamma$ for $i = 1, \cdots, n$. To solve this problem, it suffices to solve the following sub-probem. Given a workload W_0 and a certain distance α, find a workload W_1 such that $\delta(W_0, W_1) = \alpha$. Assuming we have a randomized algorithm to solve the latter problem, we can achieve the former goal by simply repeating this procedure n times, each time randomly picking a new $\alpha \in [0, \Gamma]$.

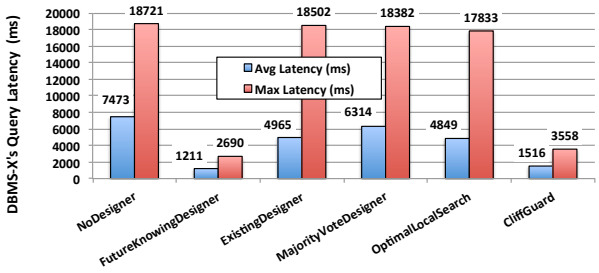
We can perform this problem efficiently when our distance metric is $\delta_{euclidean}$ defined in Section 5. The pseudocode for this procedure is presented in Algorithm 5. Here, to implement Line 2, one can construct different query sets Q by restricting oneself to only pick queries that are not already contained in W_0. One can start with $k = 1$ and if unsuccessful, continue to increase k until such a Q is found. Note that finding such a Q is much easier than the original problem, since we are not searching for a particular $\delta_{euclidean}(W_0, Q)$ distance. Rather, it suffices to find a Q that is sufficiently different from Q. Once such a Q is found, Algorithm 5 is guaranteed to find a new workload W_1 where $\delta_{euclidean}(W_0, W_1)$=$\alpha$. This can be proved as follows:

First note that $Q \cap W_0 = \emptyset$ implies that non-zero coordinates in V_{W_0} are zero in V_Q and the non-zero coordinates in V_Q are zero V_{W_0}. Thus, since $W1 \leftarrow W_0 \uplus_{i=1}^{\lfloor c \rfloor} Q$, we have:

$$V_{W_1} = \frac{n}{n + \lfloor c \rfloor \cdot k} V_{W_0} + \frac{\lfloor c \rfloor \cdot k}{n + \lfloor c \rfloor \cdot k} V_Q \qquad (12)$$



(a) Synthetic static workload S1 on DBMS-X.



(b) Synthetic drifting workload S2 on DBMS-X.

Figure 15: Avg. and worst-case performances of designers for DBMS-X, averaged over all windows.

Algorithm 5: The subroutine for sampling the workload space based on a given value of $\delta_{euclidean}$ distance.

Inputs: W_0: the initial workload,
α: the required distance
Output: W_1: a new workload satisfying $\delta_{euclidean}(W_0, W_1) = \alpha$

// Find a set of queries not contained in W_0
Find $Q = \{q_1, \cdots, q_k\}$ such that $Q \cap W_0 = \emptyset$ and $\delta_{euclidean}(W_0, Q) > \alpha$
$\beta \leftarrow \delta_{euclidean}(W_0, Q)$
$\lambda \leftarrow \sqrt{\frac{\alpha}{\beta}}$
$n \leftarrow |W_0|$ **// n is the number of queries in W_0 (including duplicate queries)**
$c \leftarrow \frac{n \cdot \lambda}{k \cdot (1-\lambda)}$ **// k is the number of iterations so far**
$W1 \leftarrow W_0 \biguplus_{i=1}^{\lfloor c \rfloor} Q$ **// take the original queries in W_0 plus $\lfloor c \rfloor$ instances of every $q \in Q$ without removing duplicates**
return W_1

Therefore, we have the following:

$$\begin{aligned}
\delta_{euclidean}(W_0, W_1) &= |V_{W_0} - V_{W_1}| \times S \times |V_{W_0} - V_{W_1}|^T \\
&= (\frac{\lfloor c \rfloor \cdot k}{n + \lfloor c \rfloor \cdot k})^2 \cdot |-V_{W_0} + V_Q| \times S \times \\
&\quad |-V_{W_0} + V_Q|^T \\
&= (\frac{\lfloor c \rfloor \cdot k}{n + \lfloor c \rfloor \cdot k})^2 \delta_{euclidean}(W_0, Q) \\
&= (\frac{\lfloor c \rfloor \cdot k}{n + \lfloor c \rfloor \cdot k})^2 \beta \approx (\frac{\frac{n \cdot \lambda}{k \cdot (1-\lambda)} \cdot k}{n + \frac{n \cdot \lambda}{k \cdot (1-\lambda)} \cdot k})^2 \beta \\
&= \lambda^2 \cdot \beta = \frac{\alpha}{\beta} \cdot \beta = \alpha \qquad (13)
\end{aligned}$$

In our experiments, we have typically found such Q with a few trials for $k \leq 5$. When c is not an integer, we use $\lfloor c \rfloor$ instances of Q to create a new workload with an integer number of different SQL queries. Thus, when $\lfloor c \rfloor \neq c$, $\delta_{euclidean}(W_0, W_1) \approx \alpha$.

D. LATENCY-AWARE DISTANCE METRICS

As discussed in Section 3, a key notion in the theory of robust optimization is incorporating uncertainty in the optimization problem formulation. Since we may not have distributional information of the uncertainty in many practical situations,[12] the uncertainty in robust optimization is expressed via an *uncertainty set*, such as a Γ-neighborhood. A major source of uncertainty in databases is that queries that are often unknown *a priori* or are subject to change.[13] Here, our parameter is our query workload, and its nominal value can be our past workload W_0. Analogously, we need to define a database-specific distance metric δ so that users can express their robustness requirements with a parameter $\Gamma \geq 0$ by demanding that their design must be robust for any future workload W as long as $\delta(W_0, W) \leq \Gamma$.

The question then is how to define a suitable distance for a pair of database workloads. To discover what makes a distance metric suitable for our database design framework, let us start by investigating the simplest choice. For ease of notation, let us assume that there are only a finite number of unique SQL queries, say N, that can be written, say $q_1, \cdots, q_N$. Then, we can represent any given workload W as $W = \{(q_i, r_i) \mid 1 \leq i \leq N\}$, where r_i is the normalized frequency of q_i in W, namely the number of queries in W that are identical to q_i divided by the total number of queries in W. Note that the same SQL query may appear several times in a workload ($r_i > 0$) and many queries may never occur in W (i.e., $r_i = 0$), but we have $\sum_{i=1}^{N} r_i = 1$.

Using this notation, the simplest notion of distance could be one that captures the changes of the frequencies of different queries between the two workloads, for instance:

$$\delta_{simple}(W_1, W_2) = \frac{1}{n} \sum_{i=1}^{N} |r_i^1 - r_i^2| \qquad (14)$$

where r_i^1 and r_i^2 are the frequencies of q_i in W_1 and W_2, respectively. Besides the obvious problem of N being too large (or infinite in reality), there is a more important problem with this δ_{simple}, which we illustrate using a toy example.

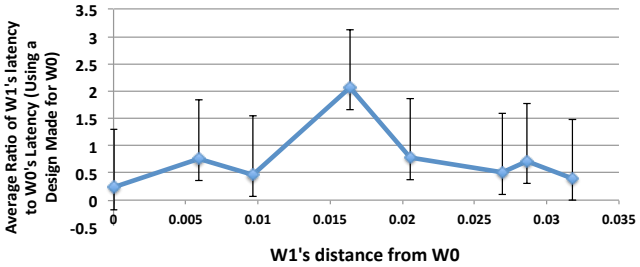
EXAMPLE D.1. *Consider a single-table database which only has 3 columns c_1, c_2 and c_3. Further assume that projection-only queries are the only type allowed in this database, i.e., each query simply projects a non-empty subset of these 3 columns without any other clauses. Below are two examples out of the $2^3 - 1 = 7$ possible query types:*

```
SELECT c1, c3 FROM T;
SELECT c1, c2, c3 FROM T;
```

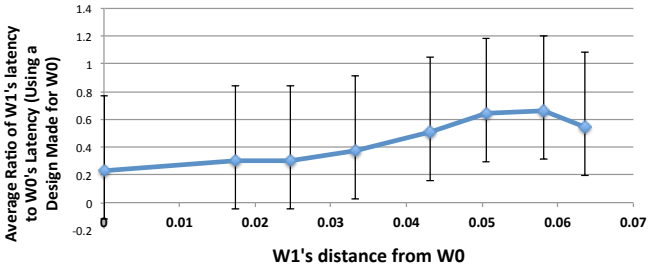
Thus, we can uniquely represent each query type as a binary string $q \in \{0,1\}^3$, where the i'th bit in q is 1 when column c_i is queried

[12] In our case, we do not know which queries will be issued with what probabilities in the future.

[13] Note that there are other sources of uncertainty in a database such as the error of our cost or resource estimates which make interesting directions for future work. In Section 6, we show that even considering workload uncertainty alone can yield substantial improvements in performance robustness.



(a) The correlation of $\delta_{latency}$ with actual performance, when $\omega = 0.1$.



(b) The correlation of $\delta_{latency}$ with actual performance, when $\omega = 0.2$.

Figure 16: Empirical evaluation of a latency-aware distance metric (compare to our $\delta_{euclidean}$ in Figure 6).

Workloads	All possible queries						
	001	**010**	**011**	**100**	**101**	**110**	**111**
W_1	0.1	0	0	0.9	0	0	0
W_2	0.15	0	0	0.85	0	0	0
W_3	0.1	0	0.05	0.85	0	0	0
W_4	0	0	0	0	0	0.5	0.5

Table 2: The frequency of different query types in four example workloads. Here, each query only projects a subset of 3 columns, represented as a binary string.

and 0 *otherwise. For example,* 101 *and* 111 *represent the two queries above, respectively. Let us consider four example workloads W_1–W_4, shown in Table 2 where each row lists the normalized frequencies of different query types in one of the workloads. E.g.,* 10% *of the queries in workload W_3 are instances of type* 001*,* 5% *are of type* 011*, and* 85% *are of type* 100*. Using Equation (14), we have*

$$\delta_{simple}(W_1, W_2) = \delta_{simple}(W_2, W_3) = \frac{0.05 + 0.05}{7} \approx 0.014$$

In other words, using δ_{simple}, W_2 is of the same distance from both W_1 and W_3. However, on a closer look at these workloads, one can see that W_2 is more similar to W_1 than it is to W_3. This is because W_1 and W_2 contain the exact same types of queries but with different frequencies, while W_1 and W_2 have different types of queries (W_2 does not contain query 011 *but W_3 does). This difference is quite important from a physical design perspective. For instance, if the database builds two materialized views for W_2 to cover both of its query types (i.e.,* 001 *and* 100*), the same views will cover all W_1 queries as well, but that may not be the case for W_3 since it contains query types not present in W_1.*

The goal of this simplified example is to show that not every distance metric is suitable for expressing the degree of robustness. Ideally, we would like a metric δ that is correlated with the performance of a physical design, namely for any workloads W_1, W_2, W_3:

$$\delta(W_1, W_2) \leq \delta(W_1, W_3) \Rightarrow f(W_2, \mathbb{D}(W_1)) \leq f(W_3, \mathbb{D}(W_1))$$

This is *soundness* criterion, presented as (8) in Section 5. Intuitively, we call a distance metric *sound*, if the smaller $\delta(W_1, W_2)$ (i.e., the more similar), the better the performance of W_2 on a design that is nominally optimal for W_1.

Note that Equation (14) can be improved by including an additive term to penalize the distance based on the number and frequency of query types that are not shared between the W_1 and W_2, i.e., they have a positive frequency in one but a zero frequency in the other. However, other limitations will remain.

For instance, the frequency differences in (14), (i.e., the $|r_i^1 - r_i^2|$ terms) only consider the frequency changes for query types individually and cannot capture the impact of frequency changes in non-identical but *similar* query types. In our Example D.1, the query type 110 is more similar to query 100 than to query 001 because a projection on columns c_1 and c_2 is likely to improve the performance of a query that only accesses c_1, but it will not help a query that accesses c_3.

Based on the assumptions and the implicit requirements of the BNT framework, we have identified the different criteria for a practical distance metric for database workloads (as presented in Section 5):

(a) The smaller the distance $\delta(W_1, W_2)$, the better the performance of W_2 on W_1's nominally optimal design, as formally stated in (8).

(b) δ should account for intra-query similarities; that is, if $r_i^1 > r_i^2$ and $r_j^1 < r_j^2$, the distance $\delta(W_1, W_2)$ should become smaller based the similarity of the queries q_i and q_j, assuming the same frequencies for the other queries.

(c) δ should be symmetric; that is, $\delta(W_1, W_2) = \delta(W_2, W_1)$ for any W_1 and W_2. (This is needed for the theoretical guarantees of the BNT framework.)

(d) δ must satisfy the *triangular property*; that is, $\delta(W_1, W_2) \leq \delta(W_1, W_3) + \delta(W_3, W_2)$ for any W_1, W_2, W_3. (This is an implicit assumption in almost all gradient-based optimization techniques, including BNT.)

As explained in Section 5, $\delta_{euclidean}$ can sufficiently capture the workload change for our physical purposes. This distance has also empirically proven both efficient and effective in our `CliffGuard` algorithm. However, we have also explored the natural question of whether incorporating performance-specific characteristics into the distance of two SQL workloads would improve our distance metric. Specifically, we have sought a more sophisticated distance metric that can satisfy the stricter requirement introduced in Section 5, namely (11), stating that for all workloads W_1, W_2, W_3 and *arbitrary* design D, δ must satisfy:

$$\begin{aligned} &\delta(W_1, W_2) \leq \delta(W_1, W_3) \quad \Rightarrow \\ &|f(W_2, D) - f(W_1, D)| \quad \leq \quad |f(W_3, D) - f(W_1, D)| \end{aligned} \tag{11}$$

In other words, the distance functions should directly match the performance characteristics of the workloads (the lower their distance, the more similar their performance).

To directly capture the strict requirement of 11, we define as latency-aware distance metric, as follows:

$$\delta_{latency}(W_1, W_2) = (1-\omega) \cdot \delta_{euclidean}(W_1, W_2) + \omega \cdot \mathcal{R}(W_1, W_2) \tag{15}$$

where $0 \leq \omega \leq 1$ is a constant and $\mathcal{R}(W_1, W_2)$ is a term capturing the difference of latency between W_1 and W_2, defined as:

$$\mathcal{R}(W_1, W_2) = \frac{|f(W_1, \emptyset) - f(W_2, \emptyset)|}{|f(W_1, \emptyset) + f(W_2, \emptyset)|} \tag{16}$$

Remember that $f(W, D)$ is the sum of the latencies of all queries in W. However, since the distance metric should be independent of a specific design, in this definition we use $D = \emptyset$, i.e., we take their baseline latencies (using no designs). In the two extreme cases, when the cost of either W_1 or W_2 is zero, $\mathcal{R}(W_1, W_2) = 1$ and when the two windows exhibit identical latencies, we have $\mathcal{R}(W_1, W_2) = 0$. The constant ω acts as a penalty factor to tune the behavior of $\delta_{latency}$, i.e., when latency difference is less important than structural similarities, one can choose a smaller value of ω (With $\omega = 0$, this distance degenerates to $\delta_{euclidean}$).

Using the same testing scenario as Section 6.3, we have empirically evaluated $\delta_{latency}$, as shown in Figure 16. Here, instead of showing the absolute latency, we have reported the ratio of the latencies of the two windows. Ideally, this ratio should be monotonic with the value of $\delta_{latency}$. This is not the case in Figure 16(a) where the penalty factor is $\omega = 0.1$. However, increasing this value to $\omega = 0.2$ yields a relatively monotonic trend, as shown in Figure 16(b). This underlines the importance of choosing an appropriate value of ω for obtaining desirable results. Our simpler distance metric $\delta_{euclidean}$, which provides a comparably reasonable accuracy in capturing the performance requirements of SQL workloads (as demonstrated in Section 6.3), does not need parameter tuning. For this and several other reasons discussed in Section 5, we currently use $\delta_{euclidean}$ in `CliffGuard`.

E. ADDITIONAL RELATED WORK

There has been a significant body of research on finding physical designs for database systems. Due to lack of space, we provide a more detailed treatment of the related work in this appendix.

While most commercial designers are based on greedy search algorithms, several academic solutions have proposed integer programming for finding the best set of views/indices. These approaches, unlike their greedy counterparts, are often guaranteed to find the (nominally) optimal design for the given workload. Even though integer programming is an NP-complete problem, most practical problem can be solved quite efficiently using the state-of-the-art solvers. By observing and characterizing various properties of the views and indices that appear in an optimal solution, Talebi et al. [79] prune the space of potentially beneficial views and indices while keeping at least one globally optimal solution in the search space. The authors also develop a heuristic procedure to further reduce the size of the search space so that the algorithm can solve larger instances of the problem. Integer programming has also been used for finding an optimal set of stratified samples in approximate databases [6, 7] and indices in PostgreSQL [64].

Lightstone et al. [62] describe Autonomic Computing (in the context of DB2 Universal Database) and enumerate the technological and manpower challenges that motivate the industrial push for self-designing, self-administering and self-tuning systems. (Autonomic computing has also been discussed in its broader context [54].) AutoAdmin [27, 31, 60] is another major research project (initiated at Microsoft over a decade ago) that focuses on automatic database tuning, including automatic selection of indices and materialized views. Li et al. [60] aims to address lack of robustness in previous approaches in resource estimation of SQL queries, by combining knowledge of database query processing with statistical models. The authors model resource usage at the level of individual operators, with different models and features for each operator type, and explicitly model the asymptotic behavior of each operator. However, our focus in this paper was on minimizing the effect of workload uncertainty on the database performance. Chaudhuri et al. [31] discuss strategies for selecting a plan with a desired tradeoff between the average and variance of query cost for different instances of parameterized queries.

LeFevre et al. [59] introduce the concept of opportunistic physical design, whereby materializing intermediate results in a MapReduce environment can lead significant opportunities for speeding up query processing (due to the overlap between revised instances of the same query). Alagiannis et al. [11] propose an interactive physical designer for PostgreSQL, which allows the DBA to simulate various physical design features and get immediate feedback on their effectiveness. The H2O system [12] dynamically adapts its data storage layout based on the incoming query workload. To determine the optimal data layout for a given workload, H2O starts with attributes accessed by the queries and progressively improves the proposed solution by considering new groups of columns. The new groups are generated by merging narrow groups with groups generated in previous iterations.

F. BASELINE ALGORITHMS

In this section, for the interested reader, we explain the baseline algorithms briefly described in Section 6.1 in further detail. As discussed earlier, the primary goal of these baselines is to evaluate the superiority of a principled approach to robust optimization (used in `CliffGuard`) over greedy heuristics and local search alternatives. The secondary goal of these baselines is to break down the contribution of various components of the `CliffGuard` algorithm to the overall performance improvement. In other words, how much of `CliffGuard`'s overall improvement over nominal designers is due to its exploration of the local neighborhood (i.e., Γ-neighborhood) of the initial workload W_0? And how much of this improvement is due to its carefully selected descent direction and step sizes in moving away from the worst neighbors? To investigate these questions, we have designed and implemented three baseline algorithms: `MajorityVoteDesigner`, `GreedyLocalSearchDesigner`, and `OptimalLocalSearchDesigner`. All three algorithms take advantage of the same neighborhood exploration strategy used in `CliffGuard` (described in Appendix C) to find perturbed workloads.

`MajorityVoteDesigner` performs sensitivity analysis to identify the *brittle* elements of the current nominal design that are more likely to be of limited benefit when the future workloads deviate from past. Given that `GreedyLocalSearchDesigner` was always inferior and significantly slower than `OptimalLocalSearchDesigner` (as explained below), we have omitted it from our experiments in Section 6. However, we still provide its detailed description for completeness.

MajorityVoteDesigner — The pseudo code of this baseline is presented in Algorithm 6. The idea behind this algorithm is to identify those elements of the design that tend to stay in the optimal design, recommended by the nominal designer, even when the workload deviates form the past. In other words, by observing how the optimal design changes with the perturbations of the current workload, `MajorityVoteDesigner` identifies the most *brittle* elements of the current nominal design and replaces them with structures that are more resilient to workload changes.

In Algorithm 6, we first sample the neighborhood of W_0 and choose n perturbed windows of distance Γ from W_0, say $W_1, \cdots, W_n$ (Line 1). Subsequently, we invoke the existing nominal designer (e.g., DBD in Vertica or DBMS-X's designer) to find the optimal

Algorithm 6: The `MajorityVoteDesigner` algorithm.

Inputs: Γ: the desired degree of robustness,
δ: a distance metric defined over pairs of workloads,
B: the (storage or maintenance) budget for building a design,
W_0: initial workload,
$\mathbb{D}$: an existing (nominal) designer,
f: the cost function (or its estimate),
Output: D^*: a design that is *hopefully* less sensitive to workload changes within W_0's Γ-neighborhood

1 $P \leftarrow \{W_i \mid 1 \leq i \leq n,\ \delta(W_i, W) \leq \Gamma\}$ **// Sample some perturbed workloads in the Γ-neighbor of W_0**
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad D_i \leftarrow \mathbb{D}(W_i)$ **// Find the nominal design for each sample neighbor**
end
5 $S \leftarrow \bigcup_{i=1}^{n} D_i$ **// Take the union of all structures (indices, materialized views, etc.) used in any of the neighbors' nominal design**
6 **for** $s \in S$ **do**
7 $\quad c_s \leftarrow \sum_{i=1}^{n} \mathbf{1}(s \in D_i)$ **// Count the number of designs that each structure s has appeared in**
end
9 $\langle s_1, s_2, \cdots, s_{|S|} \rangle \leftarrow$ sort structures in S in the decreasing order of c_s values **// s_1 appears in the most number of D_i's while $s_{|S|}$ appears the least.**
10 $D^* \leftarrow \{\}$
11 **for** $i \leftarrow 1$ **to** n **do**
12 $\quad$ **if** $price(D^* \cup \{s_i\}) \leq B$ **then**
14 $\quad\quad D^* \leftarrow D^* \cup \{s_i\}$
$\quad$ **end**
end
17 **return** D^*

Algorithm 7: The `GreedyLocalSearchDesigner` algorithm.

Inputs: Same as in Algorithm 6
Output: D^*: a design *greedily* found to minimize the cost over the entire Γ-neighborhood of W_0

1 $P \leftarrow \{W_i \mid 1 \leq i \leq n,\ \delta(W_i, W) \leq \Gamma\}$ **// Sample some perturbed workloads in the Γ-neighbor of W_0**
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad D_i \leftarrow \mathbb{D}(W_i)$ **// Find the nominal design for each sample neighbor**
end
5 $S \leftarrow \bigcup_{i=1}^{n} D_i$ **// Take the union of all structures (indices, materialized views, etc.) used in any of the neighbors' nominal design**
6 $\bar{W} \leftarrow \biguplus_{i=1}^{n} W_i$ **// Take the union of all queries from neighbors (without removing duplicates) as a representative workload**
7 $D^* \leftarrow \{\}$
8 **while** $S \neq \{\}$ **do**
10 $\quad s^* = \underset{s \in S}{ArgMin}\ f(\bar{W}, D^* \cup \{s\})$ **// Find the next best structure, given the other structures that have already been added to D^***
11 $\quad S \leftarrow S \backslash \{s\}$ **// Remove s from S so we do not re-consider it in the future**
12 $\quad$ **if** $price(D^* \cup \{s^*\}) \leq B$ ***// fits within the given budget*** **then**
13 $\quad\quad D^* \leftarrow D^* \cup \{s^*\}$
$\quad$ **end**
end
16 **return** D^*

design for each W_i, say D_i (Lines 2–3). Next, we consider the union of all physical structures (e.g., indices, materialized views) found in any of the nominal designs as our design space, called S (Line 5). This black-box treatment of the underlying nominal designer ensures that our designer remains general to arbitrary databases and physical design problems, as it does not have to construct physical structures from scratch. Rather, we simply take any structure produced by the underlying nominal designer (whether it be an index or a materialized view or a projection).[14]

To choose a subset of S, in `MajorityVoteDesigner` we count how many times each physical structure has appeared in the nominal designs of the perturbed windows (Line 6–7). We include those structures in our design that have appeared in the most number of

[14]The limitation here is that our robust designer cannot consider any structure that does not appear in the nominal designer's output for any of the perturbed windows. However, our goal is to improve on the nominal designer even without considering such structures.

Algorithm 8: The `OptimalLocalSearchDesigner` algorithm.

Inputs: Same as in Algorithm 6
Output: D^*: a design that minimizes the cost over the entire Γ-neighborhood of W_0

1 $P \leftarrow \{W_i \mid 1 \leq i \leq n,\ \delta(W_i, W) \leq \Gamma\}$ **// Sample some perturbed workloads in the Γ-neighbor of W_0**
2 **for** $i \leftarrow 1$ **to** n **do**
3 $\quad D_i \leftarrow \mathbb{D}(W_i)$ **// Find the nominal design for each sample neighbor**
end
5 $S \leftarrow \bigcup_{i=1}^{n} D_i$ **// Take the union of all structures (indices, materialized views, etc.) used in any of the neighbors' nominal design**
6 $\bar{W} \leftarrow \biguplus_{i=1}^{n} W_i$ **// Take the union of all neighbors' queries (without removing duplicates) as a representative workload**
7 $D^* \leftarrow \underset{D \subseteq S, price(D) \leq B}{ArgMin} f(\bar{W}, D)$ **// Find the best subset of the structures (e.g., using an Integer Linear Program formulation)**
8 **return** D^*

nominal designs. We start from the most popular structures (i.e., with the highest frequency count) in a greedy fashion, skipping those structures whose come exceeds our remaining (storage) budget. Thus, our strategy in `MajorityVoteDesigner` is to perform sensitivity analysis and identify those structures that are most resilient against workload changes. In other words, by counting how many times each structure appears in the neighbors' optimal designs, we select those structures that are most voted for. Structures that appear in only a few neighbors' designs are discarded as *brittle*. The idea is that such structures are less likely to be of any benefit for future workloads as they were not part of the optimal designs for most of the neighbors.

The time complexity of `MajorityVoteDesigner` is $O(n \cdot T_1 + |S|)$ where T_1 is the (average) time taken by each invocation of the nominal designer.

GreedyLocalSearchDesigner — The pseudo code for this baseline is presented in Algorithm 7. The initial steps of this baseline are identical to `MajorityVoteDesigner`, where the Γ-neighborhood of the initial workload is sampled to find n perturbed windows $W_1, \cdots, W_n$, and the union of the structures found in their nominal designs is then taken as the new design space (Lines 1–5). However, rather than taking the majority vote to choose the most frequently useful structures, `GreedyLocalSearchDesigner` uses $W_1, \cdots, W_n$ as a representative workload to evaluate the best structures. In other words, given $W_1, \cdots, W_n$ as randomly chosen neighbors, `GreedyLocalSearchDesigner` treats their combination as the *expected* future workload. To form an expected future workload $\bar{W}$, `GreedyLocalSearchDesigner` takes the union of all the queries from W_i neighbors but preserves duplicate queries in order to represent the weight of common queries in $\bar{W}$ (Line 6).

Once $\bar{W}$ is formed, `GreedyLocalSearchDesigner` starts from an empty design, and iteratively finds the best structure to add to this design until it consumes its budget (Lines 7–13). At each step, `GreedyLocalSearchDesigner` considers all pairs of the remaining structures $s, s' \in S$ and greedily decides which one is more beneficial by comparing $f(\bar{W}, D^* \cup \{s\})$ and $f(\bar{W}, D^* \cup \{s'\})$ (Line 13). This is a greedy strategy as we do not consider all subsets of structures. For instance, we may observe that $f(\bar{W}, D^* \cup \{s\}) < f(\bar{W}, D^* \cup \{s'\})$ but there may exist a different choice of D^*, say D', where $f(\bar{W}, D^*) < f(\bar{W}, D')$ but $f(\bar{W}, D^* \cup \{s\}) < f(\bar{W}, D' \cup \{s'\})$. In a greedy search, we would never discover that $D' \cup \{s'\}$ is superior to $D^* \cup \{s\}$. The next baseline (called `OptimalLocalSearchDesigner`) addresses this limitation at the cost of considering all subsets of structures.

The time complexity of `GreedyLocalSearchDesigner` is $O(n \cdot T_1 + |S|^2 \cdot T_2)$ where T_1 and T_2 are is the (average) times taken by each invocation of the nominal designer and each evaluation/estimation of the cost function f on workload $\bar{W}$, respectively. In practice, however, `OptimalLocalSearchDesigner` is significantly faster than `GreedyLocalSearchDesigner`, due to the efficiency of Integer Linear Program solvers. Moreover, the quality of `GreedyLocalSearchDesigner`'s design is by definition inferior to that of `OptimalLocalSearchDesigner`. Thus, in this paper, we have excluded `GreedyLocalSearchDesigner` from our experiments, as it is dominated by `OptimalLocalSearchDesigner`.

OptimalLocalSearchDesigner — This baseline (presented in Algorithm 8) is identical to `GreedyLocalSearchDesigner` except that instead of iteratively evaluating structures one at a time, it considers all possible subsets of the structures found in S to find a design (within budget B) which minimizes the cost over the expected workload $\bar{W}$. Since there are an exponential number of possible subsets of S, the worst-case complexity of `OptimalLocalSearchDesigner` is $O(2^{|S|})$. However, one can easily formulate such subset selection optimizations as an integer linear program (ILP) (e.g., see [6, 64]) and use the existing ILP solvers which often tend to be quite efficient in practice. For example, in all our experiments the actual time taken by GLPK (an open-source ILP solver [2]) was less than 5 seconds, even though the problem is in general NP-complete. However, before invoking the ILP solver, we need to pre-compute the cost of each query $q \in \bar{W}$ using each structure $s \in S$, namely $f(\{q\}, s)$ (which is defined as $+\inf$ if s alone is insufficient for evaluating q), leading to $O(|\bar{W}| \cdot |S|)$ invocations of the cost function. Thus, the overall time-complexity for `OptimalLocalSearchDesigner` is $O(n \cdot T_1 + |\bar{W}| \cdot |S| \cdot T_2 + T_3)$ where T_1 and T_2 are as defined above and T_3 is the time taken by the ILP solver. In practice, due to the efficiency of current ILP solvers, `OptimalLocalSearchDesigner` tends to be more efficient than `GreedyLocalSearchDesigner` (see Appendix A.4).