Variance Aware Optimization of Parameterized Queries

Surajit Chaudhuri
Microsoft Research

Hongrae Lee
University of British Columbia

Vivek Narasayya
Microsoft Research

ABSTRACT

Parameterized queries are commonly used in database applications. In a parameterized query, the same SQL statement is potentially executed multiple times with different parameter values. In today's DBMSs the query optimizer typically chooses a single execution plan that is reused for multiple instances of the same query. A key problem is that even if a plan with low *average* cost across instances is chosen, its *variance* can be high, which is undesirable in many production settings. In this paper, we describe techniques for selecting a plan that can better address the trade-off between the average and variance of cost across instances of a parameterized query. We show how to efficiently compute the skyline in the average-variance cost space. We have implemented our techniques on top of a commercial DBMS. We present experimental results on benchmark and real-world decision support queries.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *Query Processing*.

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Query optimizer, Workload, Variance

1. INTRODUCTION

Parameterized queries are widely used and play a crucial role in enterprise database applications. These include stored procedures as well as dynamic queries whose parameter values are passed in at runtime. In a parameterized query, the same SQL statement is repeatedly executed, but each instance of the query has potentially different parameter values. Consider the following query from the TPC-H benchmark [21] where @ denotes parameters.

```
SELECT   sum(l_extendedprice) / 7.0 as avg_yearly
FROM     lineitem, part
WHERE    p_partkey = l_partkey and p_size <= @size
         and p_container = @container  and p_retailprice <= @price
         and l_quantity <  ( SELECT 0.2 * avg(l_quantity)
                             FROM lineitem
                             WHERE l_partkey = p_partkey)
```

One instance of the query may be executed with parameters being bound to values (*@size=7*, *@container=*'MED BOX', *@price=3000*) whereas another instance may be executed with (*@size=6*, *@container=*'SM BOX', *@price=2000*).

SIGMOD'10, June 6-10, Indianapolis, Indiana, USA.

A key issue in plan selection for a parameterized query is identifying the criteria that the selected plan should satisfy. At one end of the spectrum is the simple approach of always compiling (i.e. optimizing) each query instance. Since each query instance is guaranteed its optimal plan, the *average* cost over all query instances is also minimized. The drawback of this approach is that it incurs significant resource overheads associated with compilation. Prior work on Parametric Query Optimization (PQO), (e.g. [2][15]), aims to select *multiple* (say *K*) plans such that the average cost over query instances is minimized. In PQO we are required to select *K* plans appropriately and also provide a method for identifying which of these plans to use for any given query instance. Finally, most commercial database systems use a simpler approach where a *single plan* is selected (i.e. *K*=1), which is then used for all query instances. The choice of this plan is once again typically optimized for average cost (e.g.[8]) over a given set of query instances.

In this paper we argue that in addition to *average* cost, a second criterion, namely the *variance* in cost over query instances is also important. The problem of using a plan with high variance is that some query instances can have good performance, but others can be unacceptably slow. This gives the impression of unpredictability in query performance, and often leads to expensive manual performance troubleshooting. Variance as an optimization criterion has received little attention in previous work on plan selection for parameterized queries. We focus on the problem of selecting a single plan for a parameterized query (a popular approach in today's DBMSs), and study it for the case when there are two optimization criteria: average and variance. Thus, our goal is to explore the trade-off between average and variance in a principled manner. For example, the DBA might be willing to use a plan whose average cost is 20% higher than the plan with the *lowest* average, but has a significantly lower variance. Note that if the DBA is able to find a plan with a suitable average-variance trade-off, then plan hinting mechanisms that are available in today's DBMSs (e.g.[9][16][20]) can be exploited to force the use of this plan for all query instances.

A useful tool for balancing multiple criteria (in this case average and variance of cost over query instances) is the *skyline*, which has proven to be valuable in multi-criteria decision making [3]. Figure 1 shows an example skyline in an average-variance plot. Each point in the plot represents a plan. The x-coordinate (resp. y-coordinate) shows the average (resp. variance) over the cost of a set of query instances when using the plan. In Figure 1, Plans 1, 3 and 6 offer interesting trade-offs between variance and average performance forming a skyline, while Plans 2, 4 and 5 are of less interest to DBAs because they are *dominated* by plans in the skyline. Thus, the ability to efficiently identify plans in the skyline of a parameterized query for a given set of instances of that query can be valuable to a DBA or application developer. We study the

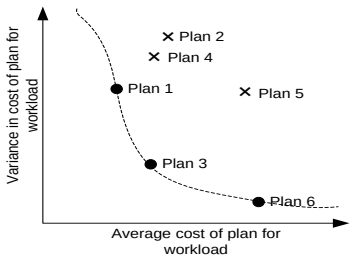


Figure 1. An example skyline of plans.

novel problem of finding the average-variance skyline of plans for a *given workload* (set of query instances). We refer to the above problem informally as the *Plan Skyline* problem.

One key challenge in computing the average-variance skyline is that the space of plans to consider for a parameterized query can be large. As shown in the previous work on *plan diagrams* [19], there could be a large number of plans that are optimal at different points in the entire selectivity space of the parameterized query. In this paper we show empirically that even the seemingly natural approach of restricting the space of plans to those that are optimal for at least one query *in the given workload* does not work well for our problem, i.e. it misses too many plans in the skyline. In other words, any plan in the plan diagram could potentially be part of the skyline. The other approach of first generating the entire plan diagram (or even an approximate plan diagram e.g. as in [11]) and then selecting the skyline plans among them does not scale when the query has many parameters. In contrast, our approach relies on sampling over the space of all plans in the plan diagram, and leverages the fact that we are not required to generate the plan diagram, but rather the plan skyline only. We establish a *probabilistic guarantee* that enables early termination by exploiting the skyline of plans observed in the sample thus far. Intuitively, this guarantee ensures that we obtain most of the skyline plans with high likelihood. We also show how to exploit workload information to bias the sampling towards "more relevant" regions of the multi-dimensional selectivity space, i.e. biasing the sampling to accelerate the termination of our skyline generation algorithm.

Another challenge is that in order to determine if a plan belongs to the skyline, we need to compute its average and variance over all query instances in the workload. The baseline approach of invoking the query optimizer to cost the plan for each query instance can be very expensive, particularly for large workloads. Thus techniques for reducing such optimizer invocations are crucial for scalability. We develop *sufficient* conditions for pruning a plan from consideration using the costs obtained for a subset of query instances in the workload. In conjunction with a carefully chosen processing order of queries, the above pruning conditions enable a significant reduction in the number of optimizer invocations.

Our solution scales well with query dimensionality (i.e. number of parameters) as well as workload size (i.e. number of query instances). We have implemented our solution on top of Microsoft SQL Server 2008, using the optimizer as a black box, and in principle can be implemented on any off-the-shelf database system. We evaluate our solution via extensive experiments on TPC-H benchmark [21] queries as well as complex decision support queries against a real world sales database. To the best of our knowledge, the problem of finding the average-variance skyline has not been studied previously. Note that despite the similarity of our problem to general skyline computation (e.g.[3]) or multi-objective optimization (e.g.[13]), the key challenges of our problem lie elsewhere, i.e. in efficiently generating and costing the plans as described above.

The rest of this paper is organized as follows. We formally define the problem in Section 2 and give an overview of the solution in Section 3. Section 4 describes our sampling based algorithm for generating the plan skyline. In Section 5 we present techniques for efficient average-variance computation using sufficient conditions for pruning. We discuss extensions to the basic algorithm in Section 6 including a generalization of the Plan Skyline problem to the case when we want to pick *K* Plans (as in the PQO work) instead of a single plan (which is the main focus of this paper). We present experimental results in Section 7 and discuss related work in Section 8.

2. PROBLEM FORMULATION

2.1 Preliminaries

In this paper, a *parameterized query* is a SQL query with placeholders for parameters in predicates. The *dimensionality* of a query is the number of parameters. For instance, the TPC-H query mentioned in the introduction is a 3-dimensional query since it has three parameters: *@size*, *@container*, and *@price*. When an instance of the query executes, the parameters are bound to specific values. We define a *workload* to be a set of instances of a given parameterized query. A workload of query instances can be collected using profiling tools in today's DBMSs, e.g. a DBA can log all instances of invocations of a particular stored procedure. Unless stated otherwise in this paper we use the term *query* to refer to an instance of a parameterized query, i.e. a query with parameter values bound.

We distinguish two types of calls to the optimizer: *optimization* and *plan costing* calls. An *optimization* call is the basic optimizer API which, given a query, returns its optimal plan. It is supported in all modern database systems. Given a query q and a plan P, which is not necessarily the optimal plan of q, a *plan costing* call returns the estimated cost of plan P for q. Optimizers that support both calls are classified as Class II optimizers in [11]. We assume a Class II optimizer which are found in several commercial database systems e.g. IBM DB2 (Optimization Profile), Microsoft SQL Server (XML Plan) and Sybase ASE (Abstract Plan).

Let the workload $W = \{q_1,\ldots,q_n\}$ be the set of instances of the parameterized query. In general each query can have an associated frequency. The techniques in this paper carry over to this case as well, so we omit this for simplicity. Given a query instance q_i and a plan P, we denote by $Cost_P(q_i)$ the optimizer estimated cost of plan P for query q_i. The average cost and the standard deviation of cost of a plan P for the workload are defined as follows.

$$avg(P) = \frac{1}{n}\sum_{i=1}^{n} Cost_P(q_i)$$

$$std(P) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Cost_P(q_i) - avg(P))^2}$$

We will denote $avg(P)$ by "*avg*" and $std(P)$ by "*std*" when P is clear or implied in the context.

Selectivity space for a parameterized query: For each parameter (i.e. dimension), the selectivity of that parameter can

take on a value between 0 and 1. Thus the entire selectivity space for a parameterized query is the space of all points in this multi-dimensional space. In practice, the selectivity space along each dimension is discretized; thus we only consider points in this discretized multi-dimensional selectivity space. This is the same approach as in the work on plan diagrams ([11][19]).

Space of plans for a parameterized query: The space of plans we consider for a given parameterized query is the union of plans that is optimal for at least one point in the selectivity space for the parameterized query (i.e. all plans in the plan diagram). We denote the space of plans for a parameterized query q by $C(q)$.

2.2 The Plan Skyline Problem

For a given workload, we say that Plan P *dominates* plan R, denoted $P < R$, if and if only $avg(P)$ and $std(P)$ for the workload are not bigger than $avg(R)$ and $std(R)$ respectively, and at least one of them is smaller than R's. In other words, P is not worse than R in either dimension, and is better in at least one of the two dimensions. We will also use $(avg(P), std(P)) < (avg(R), std(R))$ interchangeably with $P < R$.

Plan Skyline Problem: Given a parameterized query q and a workload $W = \{q_1,...,q_n\}$ consisting of instances of q, find all non-dominated plans $P \in C(q)$.

We call the set of non-dominated plans the *plan skyline*. Given the plan skyline, a DBA can choose one that appropriately balances the average cost and variance in cost.

2.3 Constrained Versions of the Problem

DBAs might also be interested in minimizing variance as long as the average performance is not "too slow". Thus, we also study constrained variations of the above Plan Skyline problem. Let the *min-avg* plan for the parameterized query q and workload W be the plan in $C(q)$ with the smallest *avg*. Likewise, define the *min-std* plan to be the one with the smallest *std*. In an *avg-std* plot (as in Figure 1), the *min-avg* plan is the leftmost plan and the *min-std* plan is the bottommost plan.

τ-Min-Var Plan Problem: Given a parameterized query q and a workload $W = \{q_1,...,q_n\}$, let S be the *min-avg* plan. Find plan P whose *std* is minimum among all plans R with $avg(R)$ being within τ % of $avg(S)$.

Intuitively, we are willing to sacrifice up to τ % of average performance compared to the plan with the best average, in order to minimize variance. Similarly, we have a dual formulation:

τ-Min-Avg Plan Problem: Given a parameterized query q and a workload $W = \{q_1,...,q_n\}$, let S be the *min-std* plan. Find plan P whose *avg* is minimum among all plans R with $std(R)$ being within τ % of $std(S)$.

We note that unlike the Plan Skyline problem that returns all plans in the skyline, the two constrained versions above return a single plan (except in the case of ties) that minimizes the measure of interest while constraining the other measure. The above formulations follow the classical approach of converting a two-criteria optimization problem into a single criterion optimization problem with a constraint on the other criterion.

3. SOLUTION OVERVIEW

3.1 Algorithm Overview

We now provide a brief overview of the algorithm we use in our solution to the Plan Skyline problem. While the structure of the algorithm (shown below) is relatively simple, each of the first three steps poses non-trivial challenges: (1) How do we know when to stop sampling (Step 1)? Intuitively, our result provides a termination condition with a *probabilistic guarantee* (configurable via an error threshold) that the skyline plans missed cannot occupy a large area in the selectivity space (Section 4). (2) Can we speed up the algorithm compared to uniform random sampling over $C(q)$ (Step 2)? We approach this using *weighted* sampling. We assign a weight to each point in the selectivity space by leveraging information about query instances in the given workload, such that plans that are likely to be in the skyline get a higher weight. We do this assignment in a scalable manner even for queries with many parameters. (3) How do we efficiently compute $avg(P)$ and $std(P)$ (as defined in Section 2.1) (Step 3)? We present sufficient conditions for pruning the *avg* and *std* computations in Section 5. Finally, updating the skyline (Step 4) is not difficult when the number of plans in the plan skyline is small, e.g. in our experiments the number of plans in the skyline was typically less than 10. Therefore, in this paper we do not focus on this step. In principle, if the number of plans in the skyline is large, previously known techniques for skyline computation (e.g.[3]) could be used.

Input: Parameterized query q, Workload W of instances of q
Output: Plan skyline for q
1. While <*termination condition not satisfied*>
2. Sample a new plan P from $C(q)$
3. Compute $avg(P)$ and $std(P)$ over query instances in W
4. Update the plan skyline appropriately

3.2 Architecture

As a proof of concept of our ideas, we have developed a Plan Skyline tool, which implements the above algorithm. Figure 2 shows an overview of how the Plan Skyline tool can be used in today's database systems.

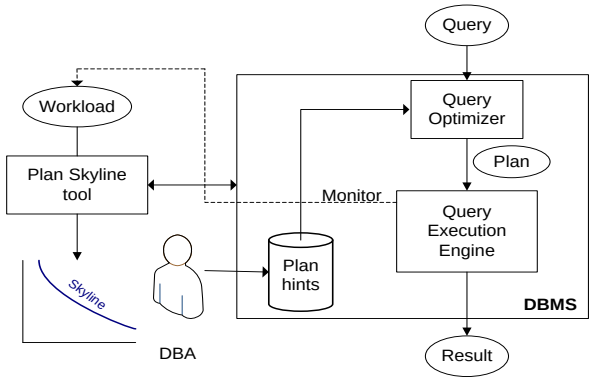


Figure 2. Architecture Overview.

The Plan Skyline tool takes as input a workload consisting of instances of a given parameterized query, and produces as output the skyline of plans (as defined in Section 2.2). We note that such a workload can be obtained using monitoring capabilities available in all commercial DBMSs, e.g. Query Patroller in IBM DB2, AWR in Oracle, and Profiler in Microsoft SQL Server. The DBA can analyze the plan skyline and select an appropriate plan among them to use for that parameterized query. Then the DBA can leverage the plan hinting mechanisms available in today's

DBMSs (e.g. [9][16][20]) to force the use of this plan for all instances of that parameterized query.

4. PLAN SKYLINE GENERATION

The Plan Skyline problem, defined in Section 2.2, requires us to find all plans $P \in C(q)$ such that P is not dominated by any other plan in $C(q)$ for the given workload W. Consider Figure 3 which shows a plan diagram [19] of a two dimensional parameterized query. For ease of exposition, we show a simplified plan diagram with rectangular regions. In reality, regions where a plan is optimal need not be rectangular or contiguous and the shape of a plan diagram can be quite complex [19]. The dots (superimposed on the plan diagram) represent query instances in workload W. Observe that there are a total of 7 plans in $C(q)$. A solution to the Plan Skyline problem, however, is required to output only the skyline plans, i.e. Plans 1, 2, 3 and 4 (the shaded plans).

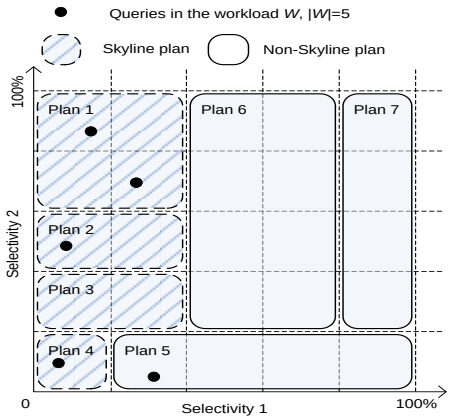


Figure 3. A plan diagram showing skyline, non-skyline plans and queries in the workload *W*.

At first glance, the Plan Skyline problem appears to be similar to the problem of efficiently generating a plan diagram for a parameterized query, i.e. generating $C(q)$. To avoid the complexity of exhaustively enumerating the entire selectivity space, the work in [11] uses an approach based on uniform random sampling of the selectivity space to identify plans. It stops when a "sufficient number" of distinct plans have been obtained. Since their goal is to generate (an approximation of) the entire plan diagram, their termination criterion is based on estimating $|C(q)|$ and stopping when the actual number of distinct plans observed via sampling is within a certain threshold of the estimated number of distinct plans.

While the above approach of sampling the selectivity space is applicable to our problem as well, there are two crucial differences. First, we need to generate the plan skyline and not $C(q)$. Since the plan skyline is typically much smaller than $C(q)$, we have an opportunity to identify a more efficient termination condition that is able to detect when we have a "sufficient number" of plans *in the plan skyline*. We present a novel method (with a probabilistic guarantee) that can tell us when we have sampled enough (Section 4.1). Second, the plan skyline is defined with respect to the *avg* and *std* over query instances in the given workload. Thus, unlike in plan diagram generation where uniform random sampling is appropriate, we are able to exploit workload information to *bias* the sampling towards regions in the selectivity space that are more likely to contain a plan in the plan skyline (Section 4.2). This can significantly speed up the algorithm.

Finally, we note that our method for sampling a *point* from the selectivity space (i.e. selectivities for each dimension) requires identifying parameter values such that the query's predicates have the required selectivities. We use an approach similar to that in [11]. Once we have a query with the desired parameter values, we invoke the query optimizer to obtain the optimal plan for that point. We omit further details due to lack of space.

4.1 Early Termination by Gap Counting

Our procedure for early termination of sampling uses the following two concepts: (a) *Region of optimality* for a given plan. This is the sub-space of the selectivity space where the given plan is optimal. (b) *Area of optimality* for a given plan. This is the total area in the selectivity space where the given plan is optimal. (Since our examples are 2-dimensional, we use the term *area*; in an n-dimensional space this represents the *volume*). In Figure 3 the selectivity along each dimension is discretized into 5 ranges, for a total of 25 cells. The region of optimality of each plan is shown in the figure as a rectangular box. The area of optimality of Plans 4 and 6 respectively are 1 and 8 cells. We emphasize that our techniques do not depend in any way on the actual shape or contiguity of the regions of optimality.

Observe that when we sample the selectivity space using *uniform random sampling*, the probability of sampling a plan is proportional to its area of optimality. Similarly, the probability of sampling any plan from a *set* of plans is simply the sum of their probabilities because the optimal regions of plans do not overlap. Consider the following example.

Example 1. In the example of Figure 3, if we sample uniformly at random, then each plan will be sampled with a probability proportional to its area. *Pr* (Plan 1) = 4/25 since Plan 1 occupies 4 cells out of 25 total cells. Likewise, *Pr* (Plan 6) = 8/25. Similarly, *Pr* (Plan 1 or Plan 2) = (4 + 2)/25 = 6/25.

Note that the above concepts generalize naturally to the case of *weighted* sampling, where the probability of sampling a point in the selectivity space is not uniform, but is proportional to a specified *weight*. Thus, in general, the probability of sampling a plan is proportional to its weight, which is the sum of the weights of all points in the selectivity space where the plan is optimal. We discuss the issue of how to assign weights based on the given workload in Section 4.2.

Probability of sampling a skyline plan: From the above discussion, it follows that the probability of sampling any plan in the plan skyline is proportional to the area (in general weights) of plans in the plan skyline.

Example 2. Assume in the example of Figure 3 that Plans 1, 2, 3 and 4 are skyline plans and the other plans are not. Then *Pr* (Sampling a skyline plan) = *Pr* (Plan 1 or Plan 2 or Plan 3 or Plan 4) = (4 + 2 + 2+ 1)/25 = 9/25.

Now consider the probability of finding a plan in the skyline, conditioned on a subset of skyline plans having already been found. The probability is proportional to the sum of the weights of *missing* skyline plans, i.e. skyline plans not yet sampled.

Example 3. *Pr* (Sampling a skyline plan | Plan 1 and Plan 2 already sampled) = *Pr* (Plan 3 or Plan 4) = 3/25. Similarly, *Pr* (Sampling a skyline plan | Plan 1, Plan 2, Plan 3 found) = *Pr* (Plan 4) = 1/25.

The above examples illustrate the key idea that the probability of finding a new skyline plan decreases as we find more skyline

plans in the sample. Thus, the more skyline plans we have already found in the sample thus far, the more samples it will take on average to find an additional skyline plan. Thus, our idea is *to stop sampling when we do not obtain a new skyline plan for a long, pre-determined gap*. We formalize this idea next.

4.1.1 Gap Counting

We maintain a counter called the *gap counter*, which keeps track of the interval between the last two new skyline plans observed *in the sample*. Each time we sample a plan, if the plan is a new plan in the skyline of all plans observed thus far in the sample, we reset the gap counter. Otherwise, we increment the gap counter.

Figure 4. A sequence of plan samples and gap counting.

Figure 4 depicts a sequence of this sampling. Circles denote "successful" sampling, i.e. the sampled plan was observed to be a new skyline plan among the plans sampled thus far. The bars denote unsuccessful sampling. Assume that after t samples, we found a successful sample. The gap counter is reset to 0 and we start counting. Let g be the number of additional samples after t to find the next successful sample. The key observation is that the probability of a successful sample is *constant* between t and $t + g$, since the sum of areas (in general weights) of the missing skyline plans does not change during this interval. If we denote the probability of a successful sample after t samples by $Pr_{ns}(t)$,

$$\forall\, 0 \le j \le g-1, Pr_{ns}(t+j) = \frac{w_{ms}}{w_{total}}$$

where w_{ms} is the weight of all missing skyline plans (i.e. skyline plans not yet sampled), and w_{total} is the total weight of all plans. The above observation implies that g follows a *geometric distribution* with a success probability of w_{ms} / w_{total}. The following theorem allows us to quantify when to stop sampling.

Theorem 1. With a probability at least $1 - \varepsilon$, if we do not find a new skyline plan in the sample for a gap $g \ge g_0 = (1 + \varepsilon^{-1/2}) / \delta$, then the sum of weights of skyline plans not yet found $\le \delta$.

Proof: See Appendix A.

Intuitively, this theorem states that if the gap counter reaches a value $\ge g_0$ (which we refer to as the *gap threshold*), then with high probability the missing skyline plans cover a small area (in general weight) in the plan diagram. For instance, suppose that we choose $\varepsilon = 0.05$ and $\delta = 0.05$, then if we do not find a new skyline plan for 110 samples, the sum of weights of missing skyline plans ≤ 0.05 with a probability at least 0.95. Note that Theorem 1 is very general in the sense that it does not depend on the query dimensionality or specific characteristics of the parameterized query. Of course, the actual sequence of gaps observed (and hence actual termination) does depend on the specific parameterized query and the workload. We observe that the weighting scheme (discussed in Section 4.2) can have a significant impact on how quickly the gap threshold is reached. For instance, consider a poor weighting scheme that assigns all skyline plans a very low weight. Then a large number of samples may be required to reach the gap threshold. The *SkylinePlans* algorithm, which implements the above gap counting technique, is shown in Figure 5.

The guarantee in Theorem 1 is probabilistic, and therefore the algorithm may miss some skyline plans. The following theorem quantifies the likelihood of missing a skyline plan based on its area of optimality.

Algorithm: SkylinePlans

Input: parameterized query Q, workload W, error threshold ε, weight sum threshold δ
Output: Set of skyline plans *Skyline*

```
1.  Skyline = φ, Plans = φ
2.  g0 = (1+ ε^-1/2) / δ
3.  g = 0
4.  While g ≤ g0
5.      Sample a point q from the selectivity space of Q
6.      Pq = optimal plan for q // optimize query q
7.      If Pq ∈ Plans      // if plan already seen
8.          g++ and continue // increment gap counter
9.      Plans = {Pq}∪ Plans   // a new plan is found
10.     Compute avg(Pq) and std(Pq) over W
11.     Remove plans dominated by Pq in Skyline
12.     If Pq is a new skyline plan
13.         g = 0 // reset the gap counter
14.         Skyline = {Pq} ∪ Skyline
15.     Else // dominated by a plan in Skyline
16.         g++   // increment gap counter
17. Return Skyline
```

Figure 5. Algorithm for generating Plan Skyline for a given parameterized query and workload.

Theorem 2. Suppose we stop sampling according to the gap threshold $g_0 = (1 + \varepsilon^{-1/2}) / \delta$ in Theorem 1. If a skyline plan P has weight $\Delta \ge \gamma \cdot \delta$, for a constant $0 < \gamma \le 1/\delta$, then the probability that plan P is not sampled is at most $e^{-\gamma(1+\varepsilon^{-1/2})}$.

Proof: See Appendix A.

Intuitively, Theorem 2 states that if a skyline plan has a "large" area of optimality (in general weight), then we will find it with high probability if we use the stopping condition of Theorem 1. Note from Theorem 2 that the probability of missing a skyline plan *decreases exponentially* with its area. For example, if $\varepsilon = 0.05$, $\delta = 0.05$, and γ=0.5 (i.e. the plan's area (weight) is at least 0.025), Theorem 2 states that the probability of not sampling the plan is at most 0.065 if we use the stopping criterion of Theorem 1. If γ=1.0 (i.e. area ≤ 0.05), the failure probability is at most 0.0043. Thus, while Theorem 1 refers to the *aggregate* weight of all missing skyline plans, Theorem 2 further shows that *individual* skyline plans with any non-trivial weight will be found with high probability.

Finally, we note that despite the stopping criterion of Theorem 1, there may exist skyline plans (with small weight) that dominate plans returned by the *SkylinePlans* algorithm. In our experimental evaluation we compare our algorithm against a method that exhaustively enumerates the discretized selectivity space (we refer to this method as *ES*) and computes the skyline among those plans. Our results (Section 7) show that: (a) We often find all skyline plans found by *ES*. (b) Even when we miss some skyline plans, in most cases the missed plans have *avg* and *std* that is very close to one or more plan returned by the algorithm. Thus, in practice, we found that the impact of the missed skyline plans is not significant.

4.1.2 The Impact of False Positives

Observe that in the above algorithm, it is possible that we think a plan P belongs to the skyline, but in reality another plan P' exists

that dominates P, but P' has not been sampled so far. We refer to such a plan P as a *false positive*. Figure 6 shows an example sequence of sampled plans. The dotted circle represents a false positive at the beginning of the gap. The false positive causes a reset of the gap counter. Thus the gap maintained by the gap counter is *shorter* than the true gap. In other words, the reset caused by a false positive can make the algorithm obtain *more* samples before we reach the gap threshold g_0. Therefore, a false positive may cause the algorithm to terminate with more samples than necessary, but never to terminate with fewer samples, which does not hurt the guarantee of Theorem 1. Finally, observe that there can never be any false negatives since a plan that is dominated by another plan in the sample itself cannot be in the true skyline.

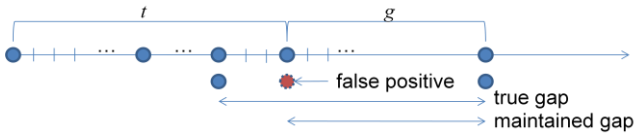


Figure 6. Impact of false positives on SkylinePlans algorithm.

4.2 Leveraging Workload for Weighted Sampling

We now describe how we assign weights to any point in the selectivity space, and thus to any plan in $C(q)$. Recollect that the weight of a plan is the sum of the weights of all points in the selectivity space where the plan is optimal. Ideally, the weight of a point should reflect the probability that the optimal plan at that point is in the plan skyline. This is however clearly infeasible since obtaining this probability would require solving the Plan Skyline problem itself. Thus, our goal is to heuristically approximate the ideal weight but at low overhead.

We develop weighting schemes based on the following two ideas: (1) A point in the selectivity space that is closer to a query instance in the workload is more likely to be the skyline plan compared to a more distant point. To motivate this, consider the case when all points in the workload have the same optimal plan P (e.g. one tight cluster of queries with high locality). In this case P would be the only point in the skyline since it would have the lowest *avg* and *std*. (2) Non-workload points have a non-zero probability of generating a skyline plan. Consider a workload with two query instances q_1 and q_2 that are *far apart* in the selectivity space. Plan P_1 is optimal for q_1 and plan P_2 is optimal for q_2. It is likely that for q_1 plan P_2 has very high cost, leading to large *avg* and *std* for P_2. Similarly since q_2 may have a high cost when using P_1, P_1 is also likely to have a large *avg* and *std*. Thus, a third plan P_3 that is not optimal for either query, but having an overall lower *avg* and *std* for q_1 and q_2 may dominate P_1 and P_2.

Our strategy is to assign a weight to a point in the selectivity space that *decreases* as the distance of that point to a point in the workload increases. The actual distance function used decides how *quickly* weight decreases as distance increases. Observe that assigning weights individually to each point in the selectivity space could be expensive since there are $O(d^k n)$ points in a k-dimensional space with d step discretization and $n = |W|$, the number of queries in the workload. Since low overhead for weight assignment is a key requirement, we consider each dimension independently. A point is projected onto each dimension and its weight on each dimension is assigned according to the projected distance between the point and query instances in the workload.

Let a^i denote point a's discretized step value in the i-th dimension. Its weight in the i-th dimension, denoted $weight_i(a)$, is defined as follows:

$$weight_i(a) = \sum_{b \in W} \frac{1}{max\ (1, dist(a^i, b^i))}$$

where *dist* is a function that measures the distance between two points. This weight can be computed efficiently by pre-computing all possible d^2 value pairs $dist(a^i,b^i)$ with $O(k(n + d^2))$ complexity. We use the *max* function to prevent division by zero.

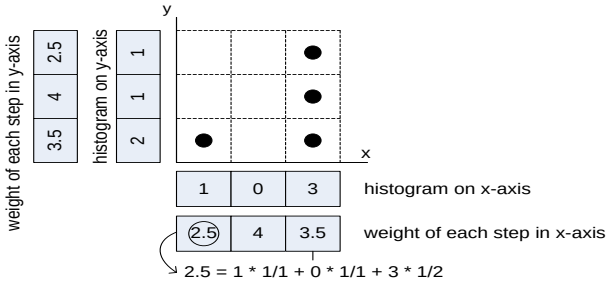


Figure 7. An example calculation of weights for a point in the selectivity space.

Example 4. Consider a 2-dimensional selectivity space where each axis is discretized into three units (see Figure 7). Suppose the workload has 4 queries**.** The counts of queries at each axis are shown as a histogram. Suppose we measure distance as the number of units between the two points. Focusing on the weight of the first step in x-axis, 1/dist values are 1/1, 1/1, and 1/2 resp. with a minimum distance of 1. Summing the values multiplied by the corresponding counts gives the weight as 2.5 (note that in practice we use normalized weights rather than absolute weights).

Sampling is done at each axis independently. Let w_i be the total weight of the i-th dimension. For example, in Figure 7, the weight of dimension x = (2.5+4+3.5) = 10. The probability of sampling a point is as follows:

$$Pr(a) = \prod_{i=1}^{d} \frac{weight_i(a)}{w_i}$$

Note that the above dimension independent weight assignment is unrelated to the independence assumption of selectivity values often made by query optimizers. The effect of independent weight assignment in our scheme is that weights can be more "spread out" and thus may lead to oversampling compared to a scheme that explicitly models correlations across dimensions.

In this paper, we evaluate the following commonly used distance measures (presented in increasing order of how quickly the weight decreases with distance):

- *Uniform*: $dist\ (a^i, b^i) = 1$
- *Sqrt*: $dist\ (a^i, b^i) = sqrt\ (|a^i - b^i|)$
- *L1*: $dist\ (a^i, b^i) = |a^i - b^i|$
- *Sq*: $dist\ (a^i, b^i) = (a^i - b^i)^2$

Our experimental evaluation (see Section 7) shows that the L1 function appears to work best: it leads to considerably less sampling by the *SkylinePlan* algorithm when compared to uniform random sampling; and it is efficient to compute.

5. EFFICIENT *AVG-STD* COMPUTATION

In this section, we present sufficient conditions for pruning the *avg-std* computation using bounding techniques. We focus on

sound pruning only, i.e. our conditions do not incorrectly prune a plan (modulo the *cost monotonicity* assumption stated below). We note that heuristic approaches such as reducing the workload size via clustering or compression (e.g. [5][12]) can also potentially be used and are orthogonal to the techniques presented here.

5.1 Lower Bounds on *Avg* and *Std*

Conceptually, the *avg-std* computation is a nested loops iteration where for each candidate plan in the outer loop we measure its costs for all query instances in the inner loop. In the context of the SkylinePlans algorithm described in Figure 5, this logic is implemented in Step 10.

1.	For each plan P
2.	For each query instance $q \in W$
3.	Get $Cost_P(q)$ by invoking the query optimizer
4.	Compute $avg(P)$ and $std(P)$.

Figure 8. Outline of *Avg-Std* computation procedure.

When we detect that plan P cannot be in the skyline after we get the cost of a query q (line 3), we prune processing P, i.e., we break out the for-loop at line 2 and move to the next plan. Our idea is to maintain lower bounds on $avg(P)$ and $std(P)$ and if we find that a plan exists in the current skyline that dominates P based on the bounds, we prune P without having to iterate over the remaining queries in the workload. Since the major cost is invoking the query optimizer, such pruning can significantly reduce the time needed. For the above bounds to be effective, we rely on the monotonic cost assumption which states that *optimizer estimated cost of the query monotonically increases in the selectivities of the query*. This assumption is observed to hold in many queries in practice and has previously been used in related contexts (e.g. [2][11]).

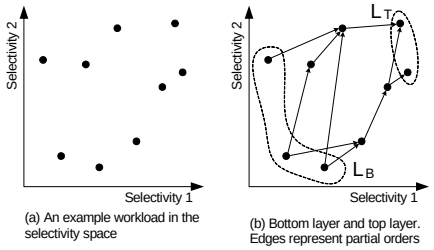


Figure 9. Top and bottom layers of queries in the workload.

Let a and b be two k-dimensional queries. They can be represented by two vectors $[a_1,\ldots,a_k]$ and $[b_1,\ldots,b_k]$ respectively in the k-dimensional selectivity space. We define a partial order $\preccurlyeq$ as follows:

$$a \preccurlyeq b \iff \forall 1 \le i \le k, a_i \le b_i$$

We say *a precedes b* if $a \preccurlyeq b$. We define the bottom layer L_B as the set of queries in workload W such that no query in W precedes them. Likewise, the top layer L_T is the set of queries that do not precede any queries in W. Note that we do not count the same queries multiple times in L_B and L_T, and a query can be in the both layers. Figure 9(a) shows an example workload in the selectivity space and 9(b) shows partial orders among queries ($a \rightarrow b$ denotes that a precedes b) and the top and the bottom layers. Not all edges are shown for clarity.

At any given time, let $min(P)$ to be the minimum cost of P over all remaining queries in the inner loop and $max(P)$ to be the maximum cost over the same remaining queries. Then, from the monotonic cost assumption:

$$min(P) = min\{Cost_P(q), q \in L_B\}$$
$$max(P) = max\{Cost_P(q), q \in L_T\}$$

Now suppose we have obtained the cost for $q_1,\ldots,q_l$ in W, where $1 < l < n$, $n = |W|$. The observation is that we can maintain a lower bound on the *avg* and *std* based on the costs seen so far, $Cost_P(q_i)$, $1 \le i \le l$. A lower bound $avg_{LB}(P,l)$ and an upper bound $avg_{UB}(P,l)$ on $avg(P)$ are as follows since costs of unseen queries $(q_{l+1},\ldots,q_n)$ are not smaller than $min(P)$ and not bigger than $max(P)$:

$$avg_{LB}(P,l) = \frac{1}{n}\left(\sum_{i=1}^{l} Cost_P(q_i) + (n-l)\cdot min(P)\right)$$
$$avg_{UB}(P,l) = \frac{1}{n}\left(\sum_{i=1}^{l} Cost_P(q_i) + (n-l)\cdot max(P)\right)$$

Similarly, we can derive a lower bound $var_{LB}(l)$ on $var(P)$ as follows:

$$var(P) = \frac{1}{n}\sum_{i=1}^{n}\left(Cost_P(q_i) - avg(P)\right)^2$$
$$= \frac{1}{n}\sum_{i=1}^{n}\left(Cost_P{}^2(q_i) - 2Cost_P(q_i)\cdot avg(P) + avg^2(P)\right)$$

$$var_{LB}(P,l) = \frac{1}{n}(X+Y) + avg_{LB}{}^2(P,l)$$

where $X = \sum_{i=1}^{l}\left(Cost_P{}^2(q_i) - 2Cost_P(q_i)\cdot avg_{UB}(P,l)\right)$

$$Y = (n-l)\cdot\left(min^2(P) - 2max(P)\cdot avg_{UB}(P,l)\right)$$

The following condition is sufficient for pruning the further processing of P where *Skyline* is the set of skyline plans so far:

$\exists$ plan $R \in Skyline$ s.t. $(avg(R), var(R)) < (avg_{LB}(P,l), var_{LB}(P,l))$

At each iteration of the inner loop we check the above condition, and if it holds, we stop processing P and move on to the next plan.

5.2 Tighter Cost Bounds

The effectiveness of the pruning conditions largely depends on how tight the cost bounds ($min(P)$ and $max(P)$) are. We describe a refinement of the above algorithm where we update the cost bounds as we process queries instead of using the bounds fixed at the start of the algorithm.

L_B (resp. L_T) in Figure 9 is in fact the skyline in the increasing (resp. decreasing) *selectivity* direction. After processing L_B and L_T, we can apply the same idea for the remaining queries repeatedly. Generalizing the top and bottom layers, we inductively define layers (a sequence of sets) of queries in W as follows:

$$L_m = \begin{cases} \left\{a : \nexists\, b \text{ s.t. } b \preccurlyeq a \text{ and } a,b \in W \setminus \bigcup_{j=1}^{m-1} L_j\right\} & \text{if } m \text{ is odd} \\ \left\{a : \nexists\, b \text{ s.t. } a \preccurlyeq b \text{ and } a,b \in W \setminus \bigcup_{j=1}^{m-1} L_j\right\} & \text{otherwise} \end{cases}$$

L_1 is the bottom layer L_B. We call L_m a *min* layer when m is odd and a *max* layer when m is even. Intuitively *min* layer L_m is the skyline in the increasing selectivity direction excluding queries in preceding layers $L_1,\ldots,L_{m-1}$. A *max* layer can be thought of in a similar way.

The Min-Max layering algorithm is straightforward, so we omit details due to lack of space. We illustrate *min-max* layering of the example workload in Figure 10. L_1 is formed by selecting nodes without incoming edges as shown in (a). We delete all associated edges after selecting the layer. Switching directions, the next layer is a *max* layer and consists of nodes without outgoing edges as in (b). We select nodes in the skyline from the remaining nodes, alternating the direction. The layering result is in (d). The algorithm runs in $O(n^2)$ time, were n is the number of queries in the workload. This preprocessing of the workload needs to be done only once at the beginning of the *SkylinePlans* algorithm. Since the cost of optimizer calls by far dominates the *SkylinePlans* algorithm, the cost of Min-Max layering is negligible.

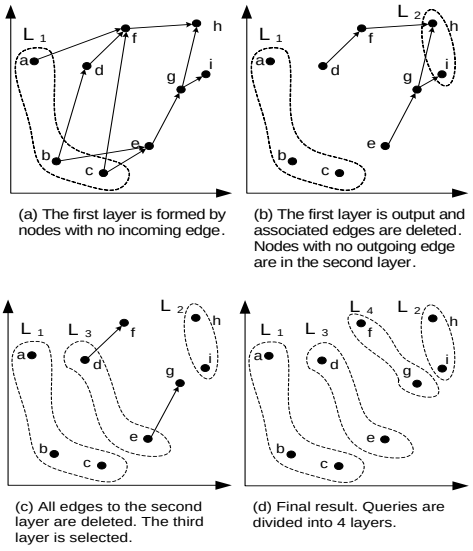


Figure 10. Min-Max layering of queries in the workload.

Our *avg-std* computation (shown in Figure 11) processes queries layer by layer in batches.

AvgStdComputation

Input: min-max layers L, plan P, min *avg* th. τ, min *avg* so far avg_{min}, current skyline *Skyline*

Output: *avg*(P) and *std*(P) or pruned

1. Process L_1 and L_2 updating *min*(P) and *max*(P) resp.
2. **For each** layer L_m, m=3,… **do**
3. **For each** query $q \in L_m$ **do**
4. Get Cost $_P(q)$ // make an optimizer call
5. Update avg_{LB} and var_{LB}
6. **If** ∃ a plan $R \in$ *Skyline*, that dominates P
7. **Return** 'P is pruned'
8. Update *min*(P) (when m is odd) or *max*(P) (otherwise)
9. Update avg_{min}
10. **Return** computed *avg*(P) and *std*(P)

Figure 11. *Avg-Std* computation with Min-Max layering.

min(P) is updated after processing a *min* layer and *max*(P) is updated after processing a *max* layer. Suppose that we processed all the queries in layer $L_1,\ldots,L_m$ and L_m is a *min* layer. *min*(P) is updated as follows:

$$min(P) = min\left\{Cost_P(q_i), q_i \in L_m \cup \bigcup_{j=1}^{\lfloor m/2 \rfloor} L_{2j}\right\}$$

min(P) is the lower bound for unseen queries in layer $L_{m+1},\ldots,L_{|L|}$.

$$min(P) \le min\left\{Cost_P(q_i), q_i \in \bigcup_{j=m+1}^{|L|} L_j\right\}$$

Note that *min*(P) is now the minimum cost of the just processed *min* layer and all the preceding *max* layers. This is because some of queries in the skyline might have been deleted in a preceding *max* layer, and thus a *min* layer is not a complete layer of skyline. *max*(P) can be defined in a similar fashion.

Finally, note that other variants of above idea of layering are also possible, e.g. have only *min* layers and thereby update only *min*(P) while fixing *max*(P) to the top layer) We empirically found that the *min-max* strategy generally worked best, and was significantly better than the basic bounding approach of Section 5.1.

6. EXTENSIONS

6.1 Handling Constraints

In Section 2.3 we presented constrained versions of the Plan Skyline problem. For example, in the τ-Min-Avg problem, we want to find the plan with the minimum variance among the plans whose average cost is within τ % of the minimum average cost. We observe that such constraint checking can be done efficiently by exploiting the bounding techniques of Section 5. In particular, we can piggyback the constraint checking onto the lower bound checking (Figure 11). We maintain avg_{min} for the minimum average cost of skyline plans so far. The following condition is a sufficient condition for pruning P.

$$avg_{LB}(P) > \left(1 + \frac{\tau}{100}\right) \cdot avg_{min}$$

Thus, after computing avg_{LB}, if the above condition is met, we stop processing P. Note however that unlike pruning by dominance (Section 5), a plan pruned by a constraint might be in the skyline. Therefore, we conservatively reset the gap counter in that case. The case of τ-Min-Std problem can be solved similarly. In our experiments (Section 7) we study the impact of constraints on the running time of the *SkylinePlans* algorithm.

6.2 Choosing Multiple Plans

Thus far our focus has been on finding a single plan that has a good trade-off between average cost and variance in cost for a given workload. Once such a plan is identified, the DBA can install that plan using a plan hint so that all instances of that parameterized query use the chosen plan. Here we briefly outline a more general version of the Plan Skyline problem that can allow the DBA to choose multiple (say K) plans. As in parametric query optimization (PQO) where using multiple plans allows reduction in average cost, using K plans here provides additional opportunities to reduce average and/or variance. Our generalization exploits the intuition of an *end-biased histogram* ([14]) which keeps K–1 "outlier" (i.e. most frequent) values separately and use one bucket to store the combined frequencies of all other values. Similarly, we can use optimal plans for K–1 "outlier" queries in the workload, and use a single plan for the rest of the queries. One advantage of such an approach is that it still easily maps to existing plan hinting mechanisms in today's DBMSs, and unlike PQO does not require specifying distance functions to map incoming query instances to one of the K plans.

While a full study of this problem is beyond the scope of this paper, below we sketch some of the issues that arise and outline potential approaches. Given a plan P, let A = $[u_1,\ldots,u_n]$ be the costs of queries in the workload with P and let B = $[v_1,\ldots,v_n]$ be

their *optimal* costs: $v_i \leq u_i$, i.e. v_i is the cost of the optimal plan for query q_i. Suppose that we use the optimal plans for some K-1 queries and use P for the rest of the queries. Let us refer to this *extended plan* as R. We observe the following: (a) The new average is never bigger than the original average: $avg(R) \leq avg(P)$. (b) In contrast, the new variance may be smaller or *bigger* than the original variance: $std(R) \nleq std(P)$. Thus, R may or may not dominate P.

There are $\binom{n}{K-1}$ possible choices of "outlier" queries for R. Thus for each plan in the original plan diagram, there are $\binom{n}{K-1}$ extended plans. We can extend the plan skyline in several ways, depending on how we choose outliers. Some choices are as follows. (1) ***Multi-plan extension***: Consider all extended plans of the original plan that are not dominated by another extended plan. (2) ***Min-Avg extension***: Among all extended plans of the original plan, choose the one with the minimum average. (3) ***Min-Var extension***: Among all extended plans of an original plan, choose the one with the minimum variance. As a proof of concept, we implemented the *Min-Avg* extension and evaluated this for different values of K. Our initial results indicate that the K-plan version of the Plan Skyline problem can significantly reduce average and variance. For example, for a workload on TPC-H Query 5 with n=100 instances, and K=10 plans, the average cost reduced by around 22% and the variance reduced by 5% compared to K=1. A full study of this problem is beyond the scope of this paper, and is an interesting area of future work.

7. EXPERIMENTS

We have implemented the techniques described in this paper on Microsoft SQL Server 2008. Below we describe the setup and results of the empirical evaluation of our techniques.

7.1 Experimental Setup

Data sets and queries: We used TPC-H benchmark [21] and a real world sales database (~10GB). We used the modified TPC-H data generator [21] to generate skewed data distributions for each column with Zipfian distribution using a skew factor of z=1. We evaluated the result on a tuned version of the 1GB, where indexes were created using recommendations from an index advisor tool. For each parameterized query, we use a workload with 100 instances for TPC-H, and 20 instances of real-world workloads for the sales database. TPC-H queries were evaluated up to 3 parameters and the real-world parameterized queries have between 2 to 6 parameters. For TPC-H queries, to obtain diverse workloads, we generate workloads using a Gaussian Mixture Model (GMM). We first generate 8 basic distributions (A ~ H) with divergent selectivity and locality as shown in Figure 12. Then two or three basic distributions are mixed to model the complex nature of real-world workloads. For example, AB is a GMM of distributions A and B.

Discretization of the selectivity space: We empirically found that finer granularity is important in small selectivity ranges (below ~2%). This is due to the fact that non-clustered indexes are typically effective in this range. Thus we use non-uniform discretization where we use finer grained intervals at small selectivity ranges. We divided the [0, 2%] range into 0.05% intervals: [0,0.05%],…,(1.95%,2%]. The rest is divided into 1% intervals: (2%,3%],…,(99%,100%].

Algorithms compared: The baseline algorithm is exhaustive search (denoted *ES*) over the full discretized selectivity space. We also implemented an algorithm (denoted *WA*) which restricts plans to only those that are optimal for at least one query instance in the given workload. In effect, this algorithm assigns non-zero weights (Section 4.2) only to points in the selectivity space corresponding to an instance in the workload. We denote the *SkylinePlans* algorithm (Figure 5) *Sky*. We set δ=0.1, ε=0.1 (see Section 4.1.1).

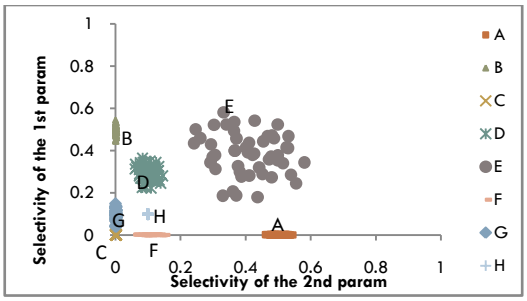


Figure 12. Synthetic workloads using Gaussian Mixture Models.

Evaluation metrics: We define the skyline recall metric to measure how effectively the algorithms find the plan skyline. It is defined as *the number of found skyline plans* / *the total number of true skyline plans*. We also show both counts. The denominator is found by an *ES* (exhaustive search). An *ES* beyond 2 parameters took too long, so we do not report the recall of real-world queries which have more than 5 parameters. For efficiency evaluation, we report the algorithm runtime. For pruning effectiveness in *avg-std* computation (Section 5), we report two figures: the fraction of pruned plans and the % of optimizer calls saved.

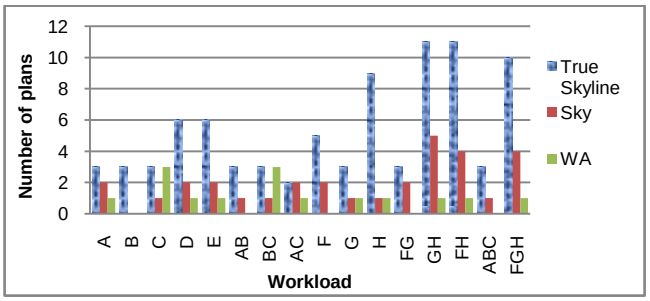


Figure 13. Comparing skyline recall of algorithms.

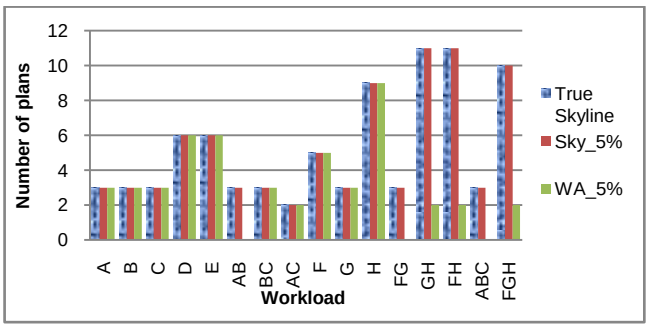


Figure 14. Skyline recall when allowing for plans with close *avg* and *std*.

7.2 Results on TPC-H

7.2.1 Comparison of algorithms

We first report the recall over 16 different workloads using Query 5 from the TPC-H benchmark. Figure 13 shows the number of skyline plans found by *Sky* and *WA* as well as the true number

found by *ES*. While both *Sky* and *WA* miss some true skyline plans, *Sky* finds more plans than *WA* for most of the workloads. At first it might appear that missing skyline plans could have an adverse affect on *avg* and *std*. However, Figure 14 shows the same experiment as above but with the following variant. If there exists a plan in the skyline found that is within 5% of both *avg* and *std* of a true skyline plan, we count the skyline as being found. We now observe that *Sky* finds all the skyline plans given this margin. This also implies that there are many skyline plans with trivial differences, which agrees with the findings in [19]. A clear contrast is that *WA* misses many skyline plans even with the margin, especially when the workload is a mix of different selectivity distributions (e.g. AB, FG). To better illustrate the above behavior, we drill-down to a specific workload AB consisting of 100 query instances. Tables 1 and 2 show respectively: (a) The distribution of optimal plans. For instance, for 23 queries in the workload, Plan 5 is optimal. (b) *Avg* and *Std* cost details for some of the plans. The table also shows if a plan is in the skyline, and whether the plan is an optimal plan for any query in the workload.

Table 1. Distribution of optimal plans for TPC-H Q5.

Optimal Plan ID	# Queries
1	3
3	11
4	13
5	23
9	50

Table 2. *Avg*, *Std* of plans for TPC-H Q5.

Plan ID	*Avg*	*Std*	Skyline?	Workload point?
12	108.60	3.21	Y	
13	108.60	3.21	Y	
11	108.60	3.21	Y	
10	109.26	1.03	Y	
7	156.67	146.96		
5	***243.46***	***235.49***		Y
25	1155.29	1150.81		
4	***1155.94***	***1151.46***		Y
3	***2030.07***	***2035.69***		Y
1	***2033.04***	***2026.88***		Y
2	5173.89	4638.06		
9	***8438.72***	***7564.61***		Y

Interestingly, compared to a skyline plan (e.g. Plan 12), Plan 5 has an *avg* that is 2x bigger and a *std* that is over 70x higher. This is because Plan 5 is optimal for certain query instances (belonging to distribution A), but performs very poorly for other instances (belonging to distribution B). Thus, *WA* which only considers plans that are optimal for at least one query in the workload will miss skyline plans such as Plan 12. Similarly plans at other workload points (e.g. Plan 1, 3, 4, 9) are also significantly worse than the skyline plans. The above example additionally illustrates how in practice (skyline) plans are often bunched together with very similar *avg* and *std* costs. This typically happens because the plans have only minor differences in the operators.

Figure 15 gives the overall skyline recall for other TPC-H queries. Some queries are excluded for the limitation of our current implementation. We observe that *Sky*'s recall is consistently above 95% while *WA*'s recall is generally between 50% and 70%.

Finally, Figure 16 shows the running time of *Sky* and *WA* for TPC-H queries. In some cases, *WA* is twice as fast as *Sky* since it typically considers much fewer plans. However, we note that *Sky* finishes reasonably quickly in absolute terms (within several minutes) whereas *ES* (not shown in the figure) typically took several *hours* for the same cases.

7.2.2 Effectiveness of pruning technique

We analyze the effectiveness of pruning conditions (Section 5) and constraints (Section 6) in reducing invocations of the query optimizer. In Figure 17, the top stack of each bar shows the number of skyline plans and the bottom stack shows the number of plans which were pruned when using the bounds on *avg* or *std* described in Section 5.2. The middle stack shows the plans that are not skyline plans but were not pruned. Figures 17 and 18 show the results when using bounds as well as a *Min-Avg* constraint (skyline plans whose *avg* is within a factor of 2 of the skyline plan with the smallest average).

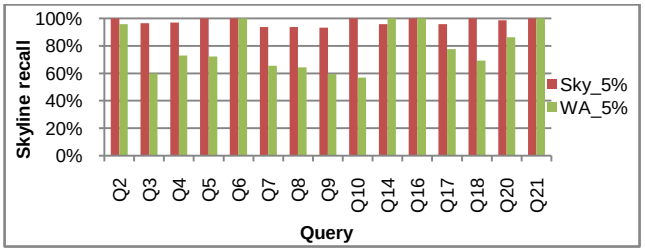


Figure 15. Sky vs. WA for TPC-H queries.

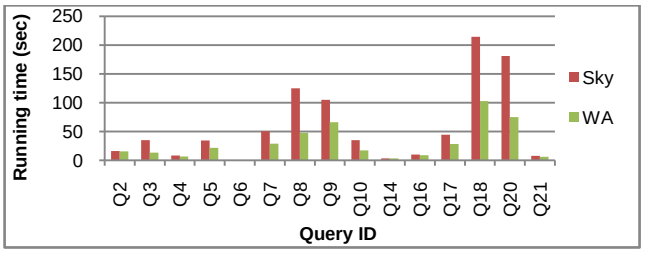


Figure 16. Running time of Sky vs. WA

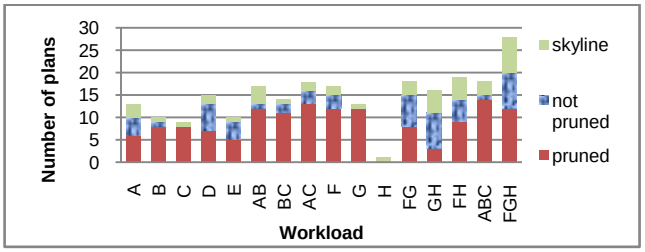


Figure 17. Pruning plans using bounds and constraints.

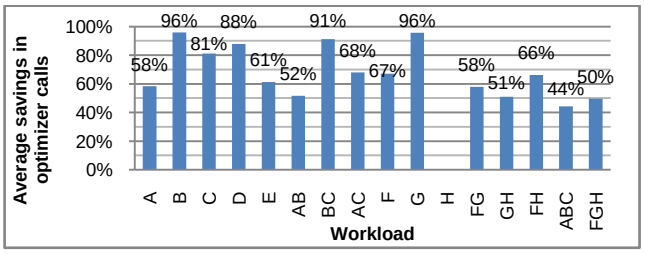


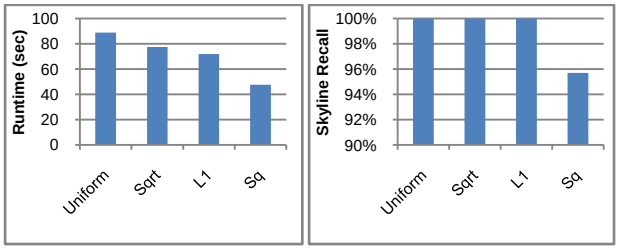
Figure 18. Optimizer calls saved by pruning.

From Figure 17 we see that pruning is effective for a large fraction of the plans (skyline plans cannot be pruned). Figure 18 shows that the pruning results in a significant savings in optimizer invocations as well. For example, a value of 91% means that for a workload with 100 queries, on average we pruned the plan after invoking the optimizer for just 9 queries. Finally, we observed significant savings in the number of plans pruned (around 50%)

and optimizer calls (around 20%) even when using only bounds (without any constraints). We omit details due to lack of space.

7.2.3 *Effect of weighting scheme*

Figure 19(a) shows the running time of various weighting schemes (described in Section 4.2) averaged over the workloads and (b) shows the skyline recall. The schemes show increasing bias towards points in the workload in the order of *Uniform*, *Sqrt*, *L1* and *Sq*. *Uniform* is the slowest since it wastes much time exploring non-skyline plans. *Sq* is the fastest since the weight decreases very quickly with distance from the workload point, but its drawback is poor skyline recall. It biases the search too much towards workload points and misses some of the skyline plans that are not from the workload. *L1* offers a good balance between the running time and skyline recall.



(a) Running time (b) Skyline recall

Figure 19. Comparing running time and recall of weighting schemes.

7.3 Results on Real World Queries

We next present the results of the real world queries in the Sales database. Unlike TPC-H, these queries include high-dimensional queries with up to 6 parameters. Below we discuss the results of a query with 5 parameters. As mentioned earlier, we were unable to run *ES* since it would have taken too long. The results were largely similar for other queries, so we omit those due to lack of space. The selectivity distribution of the 5 parameters over queries in the workload is shown in Figure 20. We notice that the selectivity distribution in the real world workload is similar to our synthetically generated distributions for TPC-H queries. Some parameters have consistently big or small selectivities while others show mixed distributions.

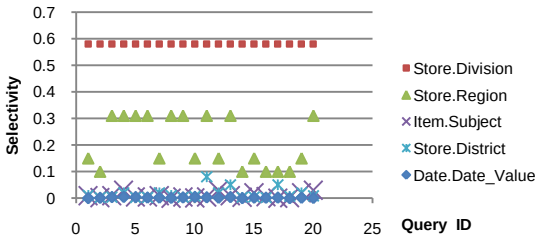


Figure 20. Selectivity distribution of parameters for a real-world query.

Table 3 shows the cost details of skyline plans. There are 4 plans in the skyline forming two clusters. We can see that the two clusters offer interesting trade-offs between average and variance.

Table 3. *Avg*, *Std* of skyline plans for a real world query.

Plan ID	*Avg*	*Std*
17	10.6	8.17
18	16.96	4.6
40	16.96	4.6
12	16.97	4.6

Plan 17 has a lower *avg* but its *std* is close to its *avg*. In contrast, the other plans have higher *avg*'s but lower *std*'s. Thus the decision of which plan to choose is not straightforward and could differ depending on the DBA's preferences. For instance, with Plan 17, query costs could vary widely between 2 and 19. A DBA may choose one from the other cluster (e.g. Plan 18) if such a big cost variance is not desirable.

Figure 21 shows diverse examples of plan skyline using TPC-H queries. The x-coordinate (resp. y-coordinate) shows the *avg* (resp. *std*) over the workload mentioned in the plot title. We observe that the same query can have very different plan skylines depending on the workload. Thus, a DBA can be presented with varying trade-offs for different cases.

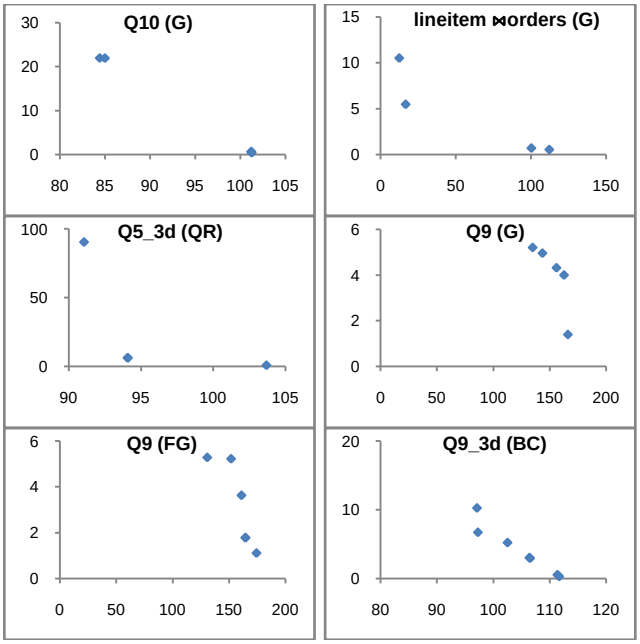


Figure 21. Diverse examples of plan skyline.

8. RELATED WORK

The concept of a plan diagram was proposed in [19] to highlight the problem in current optimizers of too many plans with only marginal cost differences. Our experimental observations agree with the claims in [11][19] and the proposed work touches upon some of the concerns in [4] by helping DBAs determine an appropriate plan to use for a parameterized query.

Least expected cost (LEC) optimization ([7][8]) aims to minimize the *expected* cost over parameter distributions. However, they do not consider variance as in our work. A brief theoretical discussion about optimizing a query for a combination of cost and variance is mentioned in [7]. Note that the algorithms in [8] are not applicable to our problem since assumptions made in the paper (such as cost linearity) does not hold for *variance* of cost unlike for the expected cost.

Parametric query optimization, (e.g. [2][15]), focuses on minimizing the average cost of the parameterized query over a given distribution of parameter/selectivity values. However, they are not concerned with minimizing variance or identifying the trade-off between average and variance cost as in our work. Robust query optimization [1] and dynamic query optimization (see [10] for an overview), are concerned with handling the *uncertainty* in selectivity estimation of input parameters *in a*

In contrast, our work considers the
ultiple instances of a parameterized qu

yline problem, although superficially s
llenges when compared with the traditi
problem [3] or the multi-objective o
. First, the points over which to compute
and are expensive to compute in our case
re-processing (e.g. indexing or sorting) is no
ints are not known a priori. Finally, unlik
kyline computation problem, space (for sky
computation) or the number points are not
our problem.

been growing interests in "predictability" in q
e.g. [17][18][22]. A distinctive aspect of our appro
rovides a clear trade-off between loss in avera
and reduction of variance. Also, our work can
complementary to the above approaches; ou
can be applied on top of any DBMS, whereas the
aches rely on changes to the query execution engine.

CLUSION

to control variance in cost of query instances of a
ed query is an important consideration in performance
wever, there is little support today for helping DBAs
suitable average vs. variance trade-off over multiple
f the query. We study this novel problem and present
and scalable solution that we have implemented on
ommercial DBMS. Our experimental results on real
benchmark queries show the promise of our techniques.
sting area of future work is to study an extension of the
lem where we are allowed to keep *multiple* plans for a
eterized query. Another orthogonal issue is to
r the functionality described in this paper can be
into the query optimizer component.

CES

vards a Robust Query Optimizer: A
Approach. *SIGMOD 2005*.
Witt, D. J. Progressive Parametric
KDE g. 21, 4 (2009), 582-594.
ocker, K. The skyline operator.

to rethink the contract?

d Summarization and

D Data

DSkew/
Query

ery

[14] Ioannidis, E. et al. Optimal histograms
error propagation in the size of join resu
[15] Ioannidis, Y. E., Ng, R. T., Shim, K., S
query optimization. *The VLDB Journal*
[16] Oracle. Using Query Hints. http://www
[17] Qiao, L. et al. Main-Memory Scan Sha
CPUs. *PVLDB 2008*.
[18] Raman, V. et al. Constant-Time Query
[19] Reddy, N., Haritsa, J. R. Analyzing Pla
Query Optimizers. *VLDB 2005*.
20] SQL Server Books Online, Hints (Trans
http://technet.microsoft.com/en-us/libra
] TPC-H Benchmark. http://www.tpc.org
Unterbrunner, U. et al. Predictable Perf
Unpredictable Workloads. In *PVLDB 2*

ndix A

f Theorem 1: Let p be the unkno
ability of sampling a new skyline
$(1\text{-}p)/p^2$. By Chebyshev's inequa

$$Pr\left[\left|g-\frac{1}{p}\right| \geq k\sqrt{\frac{1-p}{p^2}}\right] \leq$$

ound p using the above inequ

$$g^2-\frac{2g}{p}+\frac{1}{p^2} \geq k^2\frac{1-}{p}$$

$$g^2p^2-2gp+1 \geq k^2-$$

$$^2p^2+(k^2-2g)p+1-$$

Two
SH are

$$\frac{2g-k^2-k\sqrt{k^2+4g}}{2g^2}$$

$$\frac{g-k^2+k\sqrt{k^2+4g}}{2g^2}$$

Expressi
g p, p_1 and p_2,

$$p_2] \leq \frac{1}{k^2} \Rightarrow Pr[$$

Let g_o = (1
$_o$ = (1+k) / δ
lemma 1 be
(1+k) / g
inequalities,
$\geq p_2] \leq 1/k^2$
the theorem. ■

Lemma 1.

$$\frac{1}{g} <$$

Proof: We will pro
ality.

$$2g^2\left(\frac{1+}{g}\right.$$

$$= k\left(2g\right.$$

$\because (2g+k)^2 =$