Multi-Objective Parametric Query Optimization

Immanuel Trummer and Christoph Koch
Ecole Polytechnique Fédérale de Lausanne
{firstname}.{lastname}@epfl.ch

ABSTRACT

We propose a generalization of the classical database query optimization problem: multi-objective parametric query optimization (MPQ). MPQ compares alternative processing plans according to multiple execution cost metrics. It also models missing pieces of information on which plan costs depend upon as parameters. Both features are crucial to model query processing on modern data processing platforms.

MPQ generalizes previously proposed query optimization variants such as multi-objective query optimization, parametric query optimization, and traditional query optimization. We show however that the MPQ problem has different properties than prior variants and solving it requires novel methods. We present an algorithm that solves the MPQ problem and finds for a given query the set of all relevant query plans. This set contains all plans that realize optimal execution cost tradeoffs for any combination of parameter values. Our algorithm is based on dynamic programming and recursively constructs relevant query plans by combining relevant plans for query parts. We assume that all plan execution cost functions are piecewise-linear in the parameters. We use linear programming to compare alternative plans and to identify plans that are not relevant. We present a complexity analysis of our algorithm and experimentally evaluate its performance.

1. INTRODUCTION

Context. The goal of database query optimization is to map a query (describing the data to generate) to the optimal query plan (describing how to generate the data). Query optimization is a classical optimization problem with first work dating back to the seventies [14]. The original query optimization problem model has been motivated by the capabilities of data processing systems at that time. However, there have been fundamental advances in data processing techniques and systems in the meantime. Hence the original problem model is not sufficiently expressive to capture all relevant aspects of modern data processing systems. In this paper, we propose an extension of the classical query optimization problem model and a corresponding optimization algorithm.

Query optimization variants can be classified according to how they model the execution cost of a single query plan. Traditional query optimization [14] models the cost of a query plan as scalar cost value $c \in \mathbb{R}$. This implies that query plans are compared according to one single cost metric. It also implies that all information required to produce cost estimates is available to the query optimizer. The goal in classical query optimization is to find a query plan with minimal execution cost. Multi-objective query optimization [1, 7, 11, 16, 17] generalizes the classical model and associates each query plan with a cost vector $c \in \mathbb{R}^n$ instead of a scalar value. This allows to model scenarios where multiple execution cost metrics are of interest. If data processing takes place in the Cloud then we are for instance interested in execution time but also in monetary execution fees. Different components of the plan cost vector represent cost according to different cost metrics. The goal is to find the set of Pareto-optimal query plans which are plans for which no alternative plan offers better cost according to all metrics. Parametric query optimization [3, 4, 6, 8, 10, 13] generalizes the standard model in a different way. It associates each query plan with a cost function $c \in \mathbb{R}^m \to \mathbb{R}$, mapping from a multi-dimensional parameter space to a one-dimensional cost space. Parameters represent pieces of information that are not yet available at optimization time but required to estimate plan execution cost. Parametric query optimization allows for instance to optimize query classes that are defined via query templates in which some predicates are unspecified. One parameter could then represent the selectivity of one unspecified predicate. The goal in parametric query optimization is typically to find a set of plans containing for each possible parameter value combination the plan with minimal execution cost.

Problem. We propose multi-objective parametric query optimization (MPQ), a query optimization variant that generalizes multi-objective query optimization, parametric query optimization, and classical query optimization at the same time. MPQ models the cost of a single query plan as a cost function $c \in \mathbb{R}^m \to \mathbb{R}^n$ that maps a multi-dimensional parameter space to a multi-dimensional cost space. MPQ assumes that query plans are compared according to multiple cost metrics and that cost estimates depend on parameters whose values are unknown at optimization time. The goal in MPQ is to find the set of Pareto-optimal plans for each possible parameter value combination. This problem model is required wherever the application scenarios of multi-objective query optimization intersect with the ones of parametric query optimization. The following example describes a scenario in which MPQ is necessary.

EXAMPLE 1. *Assume that we need to process the same query in regular time intervals. Query processing takes place*

The original version of this article was published in the Proceedings of the VLDB Endowment, Volume 8. A video recording of the associated conference talk can be found at `http://www.itrummer.org/Talks.html`.

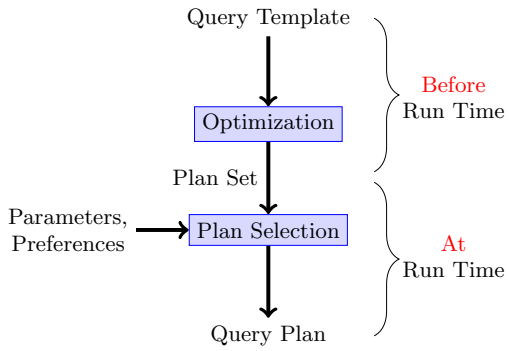

Figure 1: Multi-objective parametric query optimization pre-computes a set of relevant query plans. The optimal plan is selected from that set according to parameter values and user preferences.

in the Cloud and we would like to use Amazon EC2 Spot Instances. Here we care about two execution cost metrics which is execution time and monetary execution fees. We can trade between them by adapting type and number of the computational resources that we rent from Amazon. On the other side, the processing cost of the query depends on parameters that we cannot directly influence: the pricing of Amazon Spot Instances. As we process the same query repeatedly, we can determine the set of all potentially relevant query plans in a pre-processing step. At run time, given concrete Spot prices and execution cost bounds, we can efficiently select the best query plan out of the pre-computed set. This avoids expensive optimization at run time. The pre-processing step requires however MPQ since multiple plan cost metrics and parameters need to be considered.

There are many other scenarios in which multiple processing cost metrics are of interest. Techniques for approximate query processing allow for instance to trade between execution time and result precision [1]. Different query plans can realize different tradeoffs between energy consumption and execution time for the same query [18]. If data is processed by crowd workers then latency, execution fees, and result precision are all relevant cost metrics [12]. If the queries we want to process at run time correspond to query templates that are known before run time then we can make query optimization a pre-processing step. At pre-processing time, plan cost estimates depend on parameters with unknown values. Those parameters can represent query properties which are not fully specified in the template or properties of the query execution platform (e.g., the Spot Instance prices) that will become known only at run time. MPQ is applicable in such scenarios and allows to avoid query optimization at run time.

The result of MPQ is the set of all potentially relevant query plans for a given query or query template. It contains all Pareto-optimal plans for each possible parameter value combination. At run time, we can select the best query plan out of that set based on the concrete parameter values and based on user preferences. Users can specify their preferences in advance (e.g., by specifying cost bounds and priorities between different cost metrics [1, 16]) such that the optimal plan according to those preferences can be selected automatically. As an alternative, we can use the pre-computed plan set to visualize all Pareto-optimal cost tradeoffs for given parameter values. This allows users to select the preferred cost tradeoff directly [17]. Figure 1 illustrates the context of MPQ.

So far we have introduced a very generic problem model for MPQ. In order to make the problem tractable, we restrict ourselves to a specific class of cost functions in this paper: we consider piecewise-linear plan cost functions. Many approaches for parametric query optimization [6, 8] consider only piecewise-linear plan cost functions as well since such functions can approximate arbitrary functions [8]. The difference between our model and the one used in parametric query optimization is that we associate each plan with multiple piecewise-linear cost functions representing cost according to different metrics.

Algorithm. We present an algorithm that solves MPQ for piecewise-linear plan cost functions. Our algorithm is based on dynamic programming. It recursively decomposes the input query for which we need to determine the set of relevant query plans into sub-queries. In a bottom-up approach, it recursively calculates sets of relevant plans for a query out of optimal plan sets for its sub-queries: it combines plans that are relevant for the sub-queries to form new plans that are potentially relevant for the decomposed query. Dynamic programming is a classical approach for query optimization. The crucial difference between our algorithm and prior algorithms is the implementation of the pruning function, i.e. in how we compare alternative query plans and prune out sub-optimal plans.

We conceptually associate each plan for a query or subquery with a region in the parameter space for which the plan is Pareto-optimal. We call this region the Pareto region. The goal during pruning is to compare alternative plans generating the same result in order to discard suboptimal plans. We compare plans pair-wise and determine for each plan the parameter space region in which it is dominated by another plan, i.e. in which the other plan has comparable or better cost according to each plan cost metric. Then we reduce the Pareto region of the first plan by the region in which it is dominated. If the Pareto region of a plan becomes empty then it is not Pareto-optimal for any parameter value combination. Then we can safely discard that plan.

All Pareto regions that could ever occur during the execution of that algorithm can be represented using the following formalism. We represent Pareto regions as a union of convex polytopes in the parameter space from which other convex polytopes have been subtracted. We prove that this representation is closed under all operations that the algorithm needs to perform on Pareto regions. Note that this region shape is a consequence of the class of cost functions (piecewise-linear functions) that we consider.

The algorithm needs to perform several elementary operations on Pareto regions and cost functions. It must for instance verify whether a Pareto region is empty or calculate a parameter space region in which one plan is preferable to a second one. We show how all those operations can be implemented based on the aforementioned representation of Pareto regions. We implement those operations using linear programming.

We will formally analyze the complexity of this algorithm and present experimental results in the following sections.

Outline. The remainder of this paper is organized as follows. We define the MPQ problem and related concepts more formally in Section 2. In Section 3, we describe our algorithm for MPQ with piecewise-linear plan cost functions. In Section 4, we analyze the MPQ problem and the asymptotic complexity of our algorithm. We present experimental results for an implementation of our algorithm in Section 5 and discuss related work in Section 6.

2. FORMAL PROBLEM STATEMENT

We define the MPQ problem and related concepts more formally than in the introduction. A query describes data to generate. The description of our algorithm for solving MPQ problems, given in the next section, focuses on simple SQL join queries. An SQL join query is defined by a set of tables to join. A sub-query joins a subset of tables. Standard methods exist by which a query optimization algorithm for this simple query language can be extended into an algorithm supporting full SQL queries [14].

A query plan describes how to generate data. We say that a query plan answers a query if it generates the data that is described by the query. We assume in the following that query plans consist of a sequence of scan operations and binary join operations. For a query q, we denote by $P(q)$ the set of alternative plans that answer the query.

We compare query plans according to their execution cost. The cost of a given plan depends on a set of real-valued parameters. The set of parameters is a property of the query. All alternative plans in $P(q)$ depend therefore on the same parameters. A parameter value vector contains for each parameter a corresponding value. We do not know the parameter values at optimization time. The parameter space is the set of all possible parameter value vectors. We assume in the following that there are n parameters and denote by $X \subseteq \mathbb{R}^n$ the n-dimensional parameter space. A parameter space region is a subset of the parameter space.

We compare query plans according to multiple execution cost metrics. A cost vector contains for each cost metric a non-negative cost value. We assume in the following that there are m execution cost metrics and denote by $C = \mathbb{R}^m$ the space of cost vectors. We associate each query plan p with a cost function $c_p : X \to C$ that maps the n-dimensional parameter space to the m-dimensional cost space. We can compare the cost of query plans for specific parameter values. Denote by $x \in X$ a parameter value vector and by p_1 and p_2 two plans answering the same query. We say that p_1 dominates p_2 for x, written $p_1 \preceq_x p_2$, if p_1 has lower or equivalent cost than p_2 according to each metric for parameter values x. In other words, p_1 dominates p_2 if $c_{p_1}(x)$ contains for no component a higher value than $c_{p_2}(x)$. Now we are ready to introduce the MPQ problem.

Definition 1. An **MPQ problem** is defined by a query q, a parameter space X, and a cost space C. A solution is a subset $S \subseteq P(q)$ of query plans such that for each possible plan $p \in P(q)$ and for each possible parameter value vector $x \in X$ there is a solution plan $s \in S$ such that s dominates p for x, i.e. $s \preceq_x p$.

We focus on a sub-class of MPQ problems that restricts the class of cost functions. In order to define the class of cost functions that we consider, we must first introduce convex polytopes. A convex polytope is defined by a set of linear inequalities. The convex polytope is the set of points in the parameter space that satisfy all its inequalities. We use the terms convex polytope and polytope as synonyms. A linear cost function is defined by a constant b and an n-dimensional weight vector $w \in \mathbb{R}^n$ such that $b + w^T \times x$ is the associated cost value for each parameter vector $x \in X$. A scalar piecewise-linear cost function is a cost function that allows to partition the parameter space into convex polytopes such that the function is linear in each polytope. A vector-valued piecewise-linear cost function consists of one piecewise-linear cost function for each cost metric. We use the terms vector-valued piecewise-linear cost function and piecewise-linear cost function as synonyms. We restrict our scope to MPQ with piecewise-linear cost functions.

3. ALGORITHM

Our algorithm produces a set of relevant plans for a given query. A plan is relevant if its execution cost is Pareto-optimal for some parameter value combination.

Overview. Our algorithm splits the input query recursively into smaller and smaller parts until we obtain atomic sub-queries. We start with atomic sub-queries and calculate the set of relevant plans for each of them. After that, larger sub-queries are treated. We treat sub-queries in an order which makes sure that before treating a query, we have calculated relevant plan sets for each of its sub-queries. The reason for restricting the order is that we want to calculate the set of relevant plans for a query out of the sets of relevant plans for its sub-queries. More precisely, we can guarantee that each relevant plan for a query can be obtained by splitting the query into two sub-queries and combining a relevant plan for the first sub-query with a relevant plan for the second sub-query, thereby generating a new query plan. Having calculated the set of relevant plans for each sub-query, we can therefore obtain a superset of relevant query plans by iterating over all possible splits into sub-queries and over all possible combinations of relevant sub-plans. In order to reduce the superset to the actual set of relevant query plans, we must prune plans answering the same query. Pruning them means to identify and to discard plans that are irrelevant. The input query is treated last. The set of relevant plans for the input query is the desired algorithm output. In summary, our algorithm can be written as follows:

- Iterate over all sub-queries s of the input query in ascending order of query size:
  - If sub-query s is an atomic sub-query then consider all possible plans for s
  - Otherwise, if s is not an atomic sub-query, then iterate over all possibilities to decompose s into two sub-queries s_1 and s_2:
    - For each split into two sub-queries s_1 and s_2, consider all plans that are combinations of a relevant plan for s_1 and a relevant plan for s_2
  - Prune all considered plans to obtain the set of relevant plans for s

As many query optimization algorithms [8, 14, 16], our algorithm is based on dynamic programming. We can use dynamic programming since the principle of optimality holds for query optimization [14]. Formulated in general terms, the principle of optimality designates the problem property

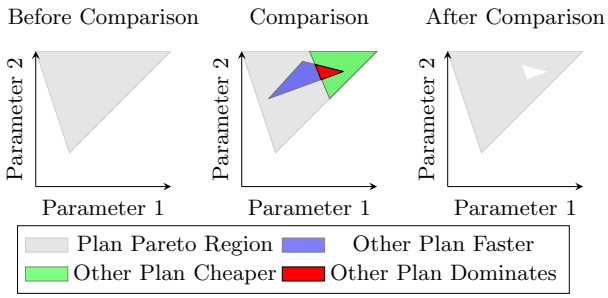

Figure 2: We subtract the area in which a plan is dominated from its Pareto region.

that optimal solutions can be obtained by combining optimal solutions to sub-problems. In the context of query optimization, the principle of optimality means more specifically that optimal query plans can be obtained by combining optimal plans for sub-queries. The principle of optimality has been shown to hold for all common execution cost metrics in multi-objective query optimization [16]. This means that a Pareto-optimal query plan can be combined from Pareto-optimal plans for sub-queries. A relevant plan is Pareto-optimal for some points in the parameter space. It is therefore intuitive that a relevant query plan can be combined from relevant plans for the sub-queries (we omit the formal proof). In other words, the principle of optimality holds for MPQ as well. It is the fundament of our MPQ algorithm.

Pruning. Many query optimization algorithms for classical query optimization [14], multi-objective query optimization [16], or parametric query optimization [8] are based on dynamic programming. The primary difference between all those algorithms is the realization of the pruning function. As we treat a novel problem variant, we must design a novel pruning function. In the following, we describe how our algorithm prunes query plans, i.e. how it compares plans for the same query and identifies irrelevant plans.

Our pruning function is based on the key concept of the Pareto region. Each query plan is associated with a Pareto region. This is a parameter space region in which it realizes Pareto-optimal cost tradeoffs. A plan is irrelevant if its Pareto region is empty. The goal of the pruning function is to compare a set of plans answering the same query in order to calculate their Pareto regions. The pruning function works as follows. At pruning start, we assume by default that each plan is Pareto-optimal in the entire parameter space. This means that we assign the entire parameter space as Pareto region to each query plan. During pruning, we compare all query plans answering the same query pair-wise in order to calculate their true Pareto regions. If we compare two plans p_1 and p_2 and we find that plan p_1 has better cost than or equivalent cost to p_2 according to all cost metrics for the parameter space region X then we reduce the Pareto region of p_2 by subtracting X. Pareto regions can only shrink during a pruning operation. Once the region of one plan becomes empty, it is irrelevant and can be safely discarded. We discard plans as soon as possible in order to avoid unnecessary comparisons.

More precisely, the pruning function iterates over all plan pairs and executes for each pair the following steps. First, it identifies the region in which one plan dominates the other plan. Second, it updates the Pareto region of the dominated plan by subtracting the region in which it is dominated. Third, it checks whether the Pareto region of the dominated plan becomes empty after the update. In that case, the plan is discarded and does not participate in further comparisons. Figure 2 illustrates how the Pareto region of a plan is reduced after comparing it to another plan. The example refers to a scenario where two parameters and two cost metrics are considered (execution time and fees).

Note that two plans can mutually dominate each other in different parameter space regions. Having determined that a first plan dominates a second plan for some parameter space region, we must therefore still verify if the second plan dominates the first plan as well.

Data Structures. We describe the data structures by which we represent plan cost functions and Pareto regions. Our plan cost model is based on piecewise-linear functions. A piecewise-linear function is linear in parameter space regions that form convex polytopes. A linear function can be represented by a constant and by weights capturing the slope of the function for each parameter. Hence a piecewise-linear function can be represented by a set of convex polytopes where each convex polytope is associated with a constant and weights. We consider multiple plan cost metrics. Each query plan is therefore associated with one piecewise-linear cost function per plan cost metric.

We consider the class of piecewise-linear cost functions to represent plan cost. We decided to use that class of functions since it allows to approximate arbitrary functions up to an arbitrary degree of precision (using more pieces increases precision). In contrast to that, we cannot freely decide which class of shapes we consider for representing Pareto regions. The algorithm must be able to represent each shape that could potentially occur during pruning. Our decision to use piecewise-linear cost functions implies the class of shapes that we need to consider as Pareto regions.

We describe our representation of Pareto regions. We motivate this representation in an informal way. It is however relatively easy to prove that the proposed representation covers all possible cases.

We start by considering the special case of linear cost functions. Parametric query optimization is a special case of MPQ. It has been shown in the domain of parametric query optimization that the parameter space region in which one plan is better than another plan according to one cost metric is a convex polytope if both plans have linear cost functions [6]. In a setting with multiple cost metrics, a plan is strictly better than another plan if it is better according to each cost metric. The region in which a plan is better than another one is therefore an intersection of multiple convex polytopes. An intersection of convex polytopes is a convex polytope again. The region in which all other plans are better than a given plan is hence a union of convex polytopes.

Now let us generalize that reasoning from linear cost functions to piecewise-linear cost functions. The generalization is straight-forward. Given two piecewise-linear cost functions, we can always partition the parameter space into convex polytopes such that both cost functions are linear in each polytope. Thereby we reduce the case of piecewise-linear cost functions to the case of linear cost functions. In summary, we can represent the Pareto region of a plan as a union of convex polytopes from which we subtract another union of convex polytopes.

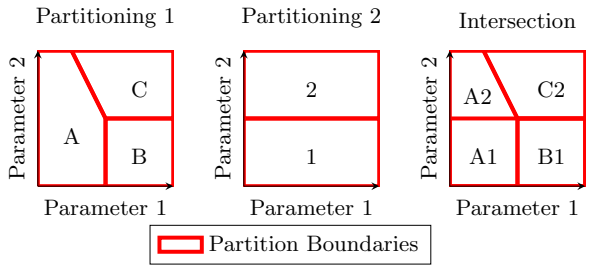

Figure 3: To compare two piecewise-linear cost functions, we intersect the parameter space partitions in which each function is linear. We compare the functions separately in each of the resulting partitions.

Elementary Operations. Having described the data structures used to represent cost functions and Pareto regions, we outline now how to implement elementary operations on those data structures. We require the following elementary operations to realize the pruning function as described before. First, given the cost functions of two plans, we must determine the parameter space region in which one plan dominates the other one. Second, given a Pareto region of a plan and a region in which it is dominated, we must reduce the Pareto region by that region. Third, given a Pareto region, we must determine whether it is empty.

Convex polytopes are described by a set of linear inequalities and we consider linear cost functions. All elementary operations that we describe in the following can hence be realized by solving systems of linear inequalities. Executing the elementary operations therefore requires a linear solver.

We describe how to determine the parameter space region in which one plan dominates another one. Assume first that we have only one cost metric and that cost functions are linear. Then we can directly use the linear solver to determine the parameter space region in which one function has lower values than the other one. Now we generalize from linear cost functions to piecewise-linear functions. Each piecewise-linear function partitions the parameter space into convex polytopes in which the function is linear. If we compare two piecewise-linear functions then we can partition the parameter space such that both functions are linear in each partition. More precisely, we obtain the aforementioned partitioning by intersecting the partitions associated with the first cost function with the partitions associated with the second function. Figure 3 illustrates how we intersect two parameter space partitionings in a two-dimensional parameter space. Having this partitioning, we apply the method for linear cost functions separately in each partition. If we have the sub-region in which a first plan dominates a second one for each parameter space region then the union of those sub-regions is the total area in which the first plan dominates. If we have multiple cost metrics instead of only one, then we can apply the method described before for each cost metric separately. If we have for each cost metric the parameter space region in which the first plan dominates the second one then the intersection of those areas (over all cost metrics) yields the area in which the first plan is better according to all cost metrics.

Given the area in which a plan is dominated, we must subtract it from that plan's Pareto region. The implementation of this operation is straight-forward: as discussed before, we represent Pareto regions as a union of convex polytopes from which other convex polytopes have been subtracted. The region in which one plan dominates another one must consist of convex polytopes. In order to subtract such a region from the Pareto region, we simply add the corresponding polytopes to the list of subtracted polytopes.

We must determine whether a given Pareto region is empty. A Pareto region is a set of polytopes from which other polytopes have been subtracted. We consider first the special case of one polytope P^+ from which another set of polytopes $\{P_i^-\}$ have been subtracted. We want to verify whether the given polytope becomes empty after the subtractions. We can verify that as follows. Assume that all subtracted polytopes P_i^- are contained within P^+. Then the region remaining after subtraction becomes empty if and only if $\cup_i P_i^- = P^+$. We can use the algorithm by Bemporad [2] to check the latter condition. The algorithm by Bemporad verifies whether the union of a given set of convex polytopes forms a convex polytope again. If this is the case then the algorithm constructs that polytope. The condition $\cup_i P_i^- = P^+$ can only be verified if $\cup_i P_i^-$ forms a convex polytope. In that case, the algorithm by Bemporad constructs the polytope $P^- = \cup_i P_i^-$ and a linear solver can verify whether P^- and P^+ are equivalent.

4. ANALYSIS

We analyze the formal properties of the freshly introduced MPQ problem in this section. We also analyze the complexity of the algorithm described in the last section.

Problem Analysis. MPQ generalizes parametric query optimization since it allows to consider multiple plan cost metrics instead of only one. We compare the formal properties of MPQ to the properties of parametric query optimization in the following.

The parametric query optimization problem with linear cost functions has the following property: if the same query plan is optimal at all vertices of a convex polytope in the parameter space then that plan must be optimal inside the polytope as well [6]. This property is commonly known as one of the "guiding principle of parametric query optimization" [5]. Many algorithms for parametric query optimization exploit this property as follows [6, 9]: they recursively decompose the parameter space into convex polytopes and calculate optimal query plans at the vertices. Due to the guiding principle, the decomposition of the parameter space can be stopped once the same plan is optimal at all vertices of a polytope. Such algorithms transform the parametric query optimization problem into a series of traditional query optimization problems (calculating the optimal plan at a polytope vertex is a traditional query optimization problem). This has the advantage that traditional query optimizers can be used for parametric query optimization with minimal changes. It is therefore interesting to verify whether an analogue property holds for MPQ.

Unfortunately this is not the case as we show next. The following property for MPQ would be analogue to the guiding principle of parametric query optimization: if the same set of plans is Pareto-optimal at all vertices of a polytope in the parameter space then that set of plans must be Pareto-optimal inside the polytope as well. Figure 4 illustrates a counter example showing that this property does not hold. The figure refers to a scenario in which two cost metrics,

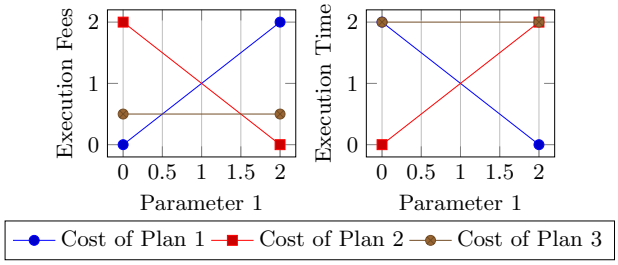

Figure 4: The guiding principle of parametric query optimization does not hold for multi-objective parametric query optimization.

namely execution time and execution fees, are of interest. Cost functions depend on a single parameter, called "Parameter 1" in the figure, that could refer to unspecified predicates in the input query template. We see the cost functions of three plans. For parameter value 0, plan 1 is Pareto-optimal since it has lowest execution fees. Plan 3 is Pareto-optimal since it has lower execution time than all other plans. Plan 2 is however dominated by plan 1 since plan 1 has equivalent execution time and lower execution cost. This means that plan 2 is not Pareto-optimal for parameter value 0. For parameter value 2, the situation is similar and plans 1 and 3 are Pareto-optimal while plan 2 is not. For parameter values between 0.5 and 1.5, plan 2 is however Pareto-optimal. Even though the same set of plans is Pareto-optimal at the borders of the parameter value interval $[0, 2]$, additional plans can be Pareto-optimal for values at the interior of that interval. All plan cost functions are linear in the example and an interval is a special case of a convex polytope. The example is minimal for MPQ: having less than two cost metrics would lead to parametric query optimization. Having less than one parameter would lead to multi-objective query optimization. Hence we can conclude from this example that the guiding principles do not apply for MPQ in general.

Algorithm Analysis. The space and time complexity of dynamic programming based query optimization algorithms depends on the number of plans stored per sub-query. In traditional query optimization, plans are compared according to one cost metric and cost functions do not depend on parameters. If we assume that alternative query plans are compared based on their cost values alone then exactly one plan, a plan with minimal cost, remains after pruning an arbitrary set of plans. In parametric query optimization, plans are compared according to one cost metric but cost functions depend on parameters. This means that different plans can be optimal for different parameter values. In multi-objective query optimization, we compare plans according to different cost metrics. Hence multiple plans can be Pareto-optimal for each sub-query. As a result, we generally need to store multiple plans per sub-query in parametric and in multi-objective query optimization. The number of plans to store depends on many factors. Research in parametric query optimization has focused on analyzing how the number of plans per sub-query depends on the number of parameters. Research in multi-objective query optimization has focused on the dependency between the number of plans and the number of cost metrics. Such analysis is necessarily based on simplifying assumptions. Traditionally, the weights that define the cost functions of different query plans are assumed to follow independent random distributions [7, 6]. Based on that assumption, the number of remaining plans after pruning can be considered a random variable as well and we can calculate its expected value. This reasoning led for instance to an asymptotic upper bound of 2^m, where m designates the number of cost metrics, on the expected number of plans per sub-query in multi-objective query optimization [7].

We perform a similar analysis to determine the expected number of plans per sub-query in MPQ. We consider linear cost functions. We denote the number of parameters by n. A linear function is therefore defined by a vector consisting of $n + 1$ components, specifying the function slope for each parameter and a constant. We still denote the number of considered plan cost metrics by m. Each query plan is therefore associated with m linear functions. The multi-dimensional cost function of each query plan can therefore be described by a matrix containing $m \cdot (n+1)$ components, specifying for each cost metric the cost slopes and a constant. Assume that we have two cost functions and that all constants and slopes describing the first function are lower than the corresponding entries for the second cost function. Then the first cost functions has for each cost metric a lower constant cost component and a lower slope in each parameter. In other words, the first cost function has lower values than the second one for arbitrary parameter values and cost metrics. If both cost functions are associated with query plans then the plan associated with the second function is clearly irrelevant.

We can exploit this fact as follows. Assume that we choose an arbitrary number of D-dimensional vectors randomly with independent identical distribution. Then the expected number of vectors such that no other vector has a lower or equivalent value in each component is bounded by 2^D [7]. We assume that vectors describing the cost functions of different query plans are chosen randomly with independent and identical distribution. Setting $D = m \cdot (n + 1)$, we infer that the expected number of vectors such that no other vector has lower or equal values in all components is bounded by $2^{m \cdot (n+1)}$. As outlined before, this is at the same time an upper bound on the expected number of relevant query plans per sub-query.

In order to obtain an upper bound on the asymptotic space complexity, we multiply the aforementioned bound by the number of sub-queries. We generate new plans by combining two relevant plans. The number of generated plans grows therefore as the square of the number of relevant plans. All generated plans for the same sub-query are compared pair-wise during pruning. The number of plan comparisons grows therefore as the fourth power of the number of relevant plans. Multiplying by the number of sub-query splits yields the time complexity measured by the number of plan comparisons.

5. EXPERIMENTS

Experimental Setup. We evaluate our MPQ algorithm experimentally. More precisely, we study how optimization time depends on the input query size and on the number of considered parameters. Our experiments are based on an example scenario in which SQL queries are processed in the Cloud. Hence we compare alternative query plans according to two cost metrics: execution time and monetary execution

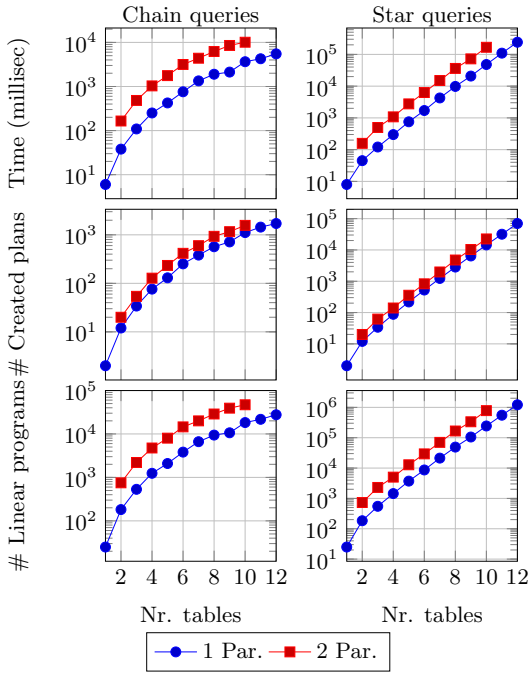

Figure 5: Optimization time, number of generated plans, and number of solved linear programs.

fees. We consider a restricted class of SQL queries: each query is described by a set of tables to join, by predicates defined on single tables, and by binary join predicates defined on table pairs. We assume that our MPQ algorithm is applied to query templates which are not fully specified: the predicates defined on single tables are placeholders. The selectivity of such a predicate, meaning the average fraction of tuples satisfying the predicate, is unknown to our MPQ algorithm. Hence the selectivity of each predicate placeholder must be represented by a parameter. Our algorithm finds all plans realizing optimal cost tradeoffs for each possible parameter value combination.

We generate the queries for our benchmark randomly. We use the method described by Steinbrunn et al. [15] to produce random queries that join a given number of tables. The number of rows in each table and the selectivity of each predicate is chosen randomly according to that method. We distinguish two classes of queries: chain queries and star queries. For chain queries, the binary join predicates connect query tables in a chain. For star queries, the binary join predicates connect one table (the middle of the "star") to all other query tables. The number of predicates is for both query classes one less than the number of tables.

We describe the plan search space that our algorithm considers. Our algorithm considers all possible orders in which tables can be joined with only one restriction: whenever we have the choice between joining two relations that are connected via a binary join predicate and joining two relations where this is not the case then only joins of the first category are considered. This restriction on the join order is often used in query optimization [14, 15]. In addition to the join orders, our algorithm considers different scan and join operators. For scanning single tables on which a predicate is defined, we consider a full scan and an index-based scan. Which of the two operators is preferable depends on the selectivity of the predicate. If the selectivity is low (few tuples will satisfy the predicate) then the index scan is often preferable. If the predicate is satisfied for most tuples then the full scan is more efficient. We model the selectivity of a predicate defined on a single table by a parameter. The optimal choice for the scan operator therefore depends on the value of that parameter. We consider two join operators: a distributed join and a single-node hash join. For sufficiently large amounts of input data, the distributed join saves execution time. On the other side, the distributed join requires to rent more computational resources from the Cloud provider and is therefore more expensive. Hence we can realize different tradeoffs between execution time and execution fees by selecting between alternative join operators. We implemented our MPQ algorithm in Java 1.7. We used Gurobi 5.6 as linear solver. All experiments were executed on an iMac equipped with an i5-3470S processor with 2.9 GhZ and 16 GB of RAM.

Experimental Results. Figure 5 shows our experimental results. Each data point in that figure corresponds to the median value of 25 randomly generated test cases. We report optimization time, the number of generated query plans (counting plans for the input query and plans for subqueries), and the number of solved linear programs. We generated query templates joining between two and 12 tables and having between one and two parameters.

Optimization time increases in the number of tables. As predicted by our formal analysis in the previous section, optimization time also increases in the number of parameters. Optimization time grows faster in the number of query tables for star queries than for chain queries. The reason is that the number of admissible join orders grows faster in the number of query tables for star queries. Speaking of admissible join orders, we mean join orders that comply with the restriction mentioned before. Optimization time, the number of generated plans, and the number of solved linear programs are all correlated. This is intuitive as the number of generated plans relates to the number of plan comparisons that are required during pruning. The number of linear programs is related to the number of plan comparisons since plan comparisons are realized by solving linear programs. The time required for generating plans and for solving linear programs adds to optimization time.

The query sizes that we consider in our benchmark are typical for query sizes as they appear in standard benchmarks: the queries in the popular TPC-H benchmark join for instance at most eight tables. MPQ takes longer than traditional query optimization. In contrast to traditional query optimization, MPQ takes however place before run time. This makes higher optimization times acceptable.

6. RELATED WORK

Figure 6 shows how multi-objective parametric query optimization relates to prior query optimization variants. The figure shows for each variant the type of cost function c that is associated with each query plan. Arrows point from a more restricted to a more general query optimization vari-

`http://www.gurobi.com/`
`http://www.tpc.org/tpch/`

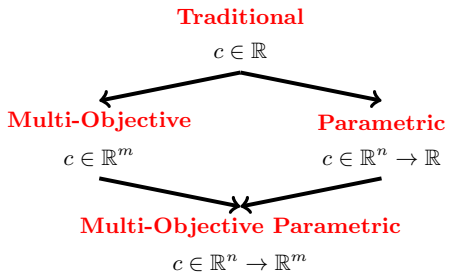

Figure 6: Multi-objective parametric query optimization generalizes the cost models of multi-objective and of parametric query optimization.

ant. Multi-objective query optimization [1, 7, 11, 16, 17] and parametric query optimization [3, 4, 6, 8, 10, 13] both generalize the traditional query optimization model [14]. Multi-objective parametric query optimization generalizes both of the aforementioned variants.

The algorithm that we propose in this paper allows to solve query optimization problems that prior algorithms cannot solve. Algorithms for parametric query optimization are not applicable to MPQ since they cannot handle multiple cost metrics. Algorithms for mult-objective query optimization are not applicable to MPQ since they cannot handle parameters. Note that parameters and cost metrics have a different semantic such that it is not possible to model parameters as cost metrics or vice versa. Intuitively, we want to "cover" the entire parameter space (by finding plans for each possible parameter value combination) while we do not want to cover the entire cost space (plans with higher cost values than necessary are not part of the result plan set).

The algorithm that we describe in this article is based on dynamic programming. It calculates optimal plans for a query by combining optimal plans for its sub-queries. Many query optimization algorithms for traditional query optimization [14], multi-objective query optimization [16, 17], and parametric query optimization [8] use the same dynamic programming scheme. The difference between our algorithm and all prior algorithms lies in the implementation of the pruning function. We use linear programming in the pruning function. Our algorithm shares this property with prior algorithms for parametric query optimization [8]. We support however multiple cost metrics and hence the definition of the pruning function, the type of the used data structures, and the implementation of elementary operations on those data structures differ.

Many algorithms for parametric query optimization are based on the guiding principles of parametric query optimization [5]. They partition the parameter space in a more and more fine-grained manner until a single query plan is optimal in each partition [6, 9]. The condition that allows to verify whether a single query plan is optimal in a given partition is based on the guiding principles. We have shown in Section 4 that the multi-objective analogue of the guiding principles does not hold for MPQ. Hence we cannot use generalizations of the aforementioned decomposition methods for MPQ.

7. CONCLUSION

The traditional query optimization model is outdated. We proposed a generalized problem model that allows to represent multiple plan cost metrics and multiple parameters. We described and analyzed a first algorithm that solves the novel optimization problem.

8. REFERENCES

[1] S. Agarwal, A. Iyer, and A. Panda. Blink and it's done: interactive queries on very large data. In *VLDB*, volume 5, pages 1902–1905, 2012.

[2] A. Bemporad, K. Fukuda, and F. Torrisi. Convexity recognition of the union of polyhedra. *Computational Geometry*, 18(3):141–154, 2001.

[3] P. Bizarro, N. Bruno, and D. DeWitt. Progressive parametric query optimization. *KDE*, 21(4):582–594, 2009.

[4] P. Darera and J. Haritsa. On the production of anorexic plan diagrams. *PVLDB*, 2007.

[5] A. Dey, S. Bhaumik, and J. Haritsa. Efficiently approximating query optimizer plan diagrams. In *VLDB*, pages 1325–1336, 2008.

[6] S. Ganguly. Design and analysis of parametric query optimization algorithms. In *VLDB*, pages 228–238, 1998.

[7] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, pages 9–18, 1992.

[8] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB*, pages 167–178, 2002.

[9] A. Hulgeri and S. Sudarshan. AniPQO: almost non-intrusive parametric query optimization for nonlinear cost functions. In *PVLDB*, pages 766–777, 2003.

[10] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric Query Optimization. *VLDBJ*, 6(2):132–151, may 1997.

[11] C. Papadimitriou and M. Yannakakis. Multiobjective query optimization. In *PODS*, pages 52–59, 2001.

[12] H. Park and J. Widom. Query optimization over crowdsourced data. *VLDB*, pages 781–792, 2013.

[13] N. Reddy and J. Haritsa. Analyzing plan diagrams of database query optimizers. *VLDB*, pages 1228–1239, 2005.

[14] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[15] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDBJ*, 6(3):191–208, aug 1997.

[16] I. Trummer and C. Koch. Approximation schemes for many-objective query optimization. In *SIGMOD*, pages 1299–1310, 2014.

[17] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In *SIGMOD*, pages 1941–1953, 2015.

[18] Z. Xu, Y. C. Tu, and X. Wang. PET: Reducing Database Energy Cost via Query Optimization. *VLDB*, 5(12):1954–1957, 2012.