An Incremental Anytime Algorithm for Multi-Objective Query Optimization*

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

ABSTRACT

Query plans offer diverse tradeoffs between conflicting cost metrics such as execution time, energy consumption, or execution fees in a multi-objective scenario. It is convenient for users to choose the desired cost tradeoff in an interactive process, dynamically adding constraints and finally selecting the best plan based on a continuously refined visualization of optimal cost tradeoffs. Multi-objective query optimization (MOQO) algorithms must possess specific properties to support such an interactive process: First, they must be anytime algorithms, generating multiple result plan sets of increasing quality with low latency between consecutive results. Second, they must be incremental, meaning that they avoid regenerating query plans when being invoked several times for the same query but with slightly different user constraints. We present an incremental anytime algorithm for MOQO, analyze its complexity and show that it offers an attractive tradeoff between result update frequency, single invocation time complexity, and amortized time over multiple invocations. Those properties make it suitable to be used within an interactive query optimization process. We evaluate the algorithm in comparison with prior work on TPC-H queries; our implementation is based on the Postgres database management system.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

Query optimization; multi-objective; incremental; anytime

1. INTRODUCTION

Classical query optimization considers only one cost metric for query plans and aims at finding a plan with minimal cost [12]. This model is insufficient for scenarios where multiple cost metrics are of interest. Multi-Objective Query Optimization (MOQO) judges query plans based on multiple cost metrics such as monetary fees of execution (e.g., in cloud computing) and energy consumption in addition to execution time [11, 14, 15]. Plans are associated with cost vectors instead of cost values and the goal is to find a plan with an optimal tradeoff between conflicting cost metrics. The optimal tradeoff is user-specific since different users might have different priorities. Prior work in MOQO assumes that users select the optimal cost tradeoff indirectly by specifying weights and constraints prior to query optimization. User studies have however shown that users have generally troubles to accurately express their preferences indirectly in a multi-objective scenario without having prior knowledge of the available tradeoffs [17]. It is more natural for users to select the preferred tradeoff out of a set of alternatives and this procedure tends to lead users to better choices. We apply those results from general multi-objective optimization to MOQO and postulate that MOQO should be an interactive process (at least for queries with non-negligible execution time) in which users select the query plan with optimal cost tradeoff out of a set of alternatives. The following examples illustrate two out of many possible application scenarios.

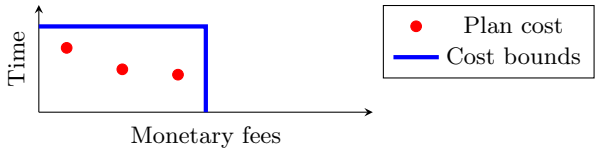
EXAMPLE 1. *In cloud computing, there is a tradeoff between execution time and fees as buying more resources can speed up execution. Users performing SQL processing in the cloud can benefit from a visualization of available cost tradeoffs before they select a query plan for execution.*

EXAMPLE 2. *In approximate query processing, there is a tradeoff between execution time and result precision since sampling can be used to reduce execution time. Visualizing available tradeoffs helps users to hand-tune the execution of queries that process large data sets or are executed frequently.*

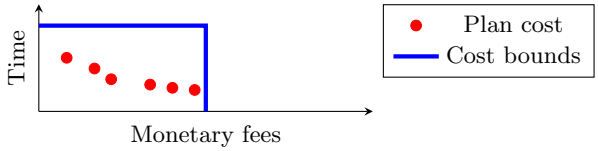
It is not necessary to make users aware of all alternative query plans for their query. It is sufficient if users have an overview of the *Pareto-Optimal* plans; a plan is Pareto-optimal if no alternative plan has better cost according to all cost metrics at the same time (this definition is slightly simplified). For two or three cost metrics, the Pareto-optimal plan cost tradeoffs can be visualized as a curve or as a surface in three-dimensional space. For higher number of cost metrics, users could successively visualize the Pareto surface for different combinations of cost metrics or look at aggregates (minima and maxima) for the different cost metrics. Having an overview of the available cost tradeoffs, users can directly select the query plan which fits best to their priorities.

An ideal interactive MOQO optimizer presents an overview of all Pareto-optimal cost tradeoffs quickly after receiving

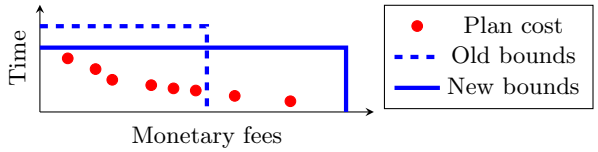
*This work was supported by ERC grant 279804.



(a) The optimizer quickly visualizes a first approximation of Pareto-optimal query plan costs



(b) The approximation is continuously refined without user interaction

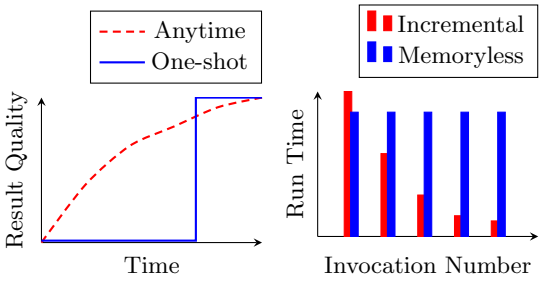


(c) The user can always adapt the optimization focus by dragging bounds into a new position

Figure 1: Example interaction between user and interactive anytime optimizer: the user selects a query plan by finally clicking on the desired cost tradeoff.

the user query as input. The problem is that the number of Pareto plans might be extremely large already for medium-sized queries and prior work has shown that calculating all Pareto-optimal plans is often not realistic within a reasonable time frame [14]. This leads to approximation algorithms for MOQO that quickly find a representative set of query plans whose cost vectors approximate the Pareto-optimal cost tradeoffs with a given target precision [11, 14]. There is a tradeoff between optimization time and target precision; choosing a finer target precision increases optimization time. Approximate MOQO can take several tens of seconds for TPC-H queries when choosing a rather fine-grained target precision which is inconvenient for an interactive interface. It is impossible to know which precision the user requires to make his decision. It is also unclear how much time optimization will take for a given query and target precision since this depends on the size of the result plan set which is the output of optimization. The most natural approach is therefore an interface that iteratively refines the approximation of the Pareto cost tradeoffs, while allowing continuous user interaction; users may for instance interact with the MOQO optimizer by selecting a query plan for execution (thereby ending optimization) or by setting cost bounds for different cost metrics (which can be exploited to speed up optimization as bounds restrict the search space). Figure 1 illustrates the interaction between user and optimizer: query plans are evaluated according to the two cost metrics (execution) time and monetary fees in the example, and plan costs are represented as points in a two-dimensional space.

Existing approximation algorithms for MOQO [14] are however ill-suited to be used within such an interface for several reasons. First, they require to specify a target precision in advance and return results only once a full result



(a) Anytime versus one-shot algorithms

(b) Incremental versus memoryless algorithms

Figure 2: Incremental anytime algorithms

plan set is generated that is guaranteed to approximate the Pareto plan set with the target precision. An interactive scenario rather requires algorithms that return several result plan sets of increasing approximation precision with high frequency (low waiting time between consecutive result sets). Algorithms that continuously improve result quality instead of returning only one result at the end of execution are generally called *anytime algorithms* in contrast to *one-shot algorithms*. Figure 2(a) illustrates the difference. A second shortcoming of existing MOQO algorithms is that they are *non-incremental*: they cannot systematically exploit results of prior invocations to speed up optimization for very similar input problems. Users might for instance adapt the cost bounds several times when optimizing a single query which changes part of the input for the optimization algorithm (the bounds change while the query remains the same). Starting optimization from scratch every time that this happens is inefficient since the same query plans might get regenerated several times. An interactive scenario rather requires an *incremental* algorithm that maintains state across several invocations for the same query to minimize redundant computation. Figure 2(b) illustrates the difference between incremental and memoryless algorithms.

The original scientific contribution of this paper is an *incremental anytime algorithm for MOQO*. This algorithm has the anytime property since it generates a rough approximation of the Pareto plan set quickly that is refined in multiple steps, having low latency between consecutive refinements. The algorithm is incremental since it maintains state across consecutive invocations for the same query with different cost bounds, thereby avoiding to regenerate the same plans. Hence our algorithm is suitable for interactive MOQO.

We summarize the contributions of this paper:

1. We present an incremental anytime algorithm for interactive MOQO; it continuously refines the approximation of the Pareto-optimal cost tradeoffs and avoids regenerating plans over multiple invocations.

2. We analyze the space complexity of that algorithm, the time complexity of a single invocation, and the amortized complexity of several invocations.

3. We experimentally evaluate an implementation of that algorithm within the Postgres database management system comparing with non-incremental non-anytime MOQO algorithms, using TPC-H queries as test cases.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces the formal model that is used throughout the remainder of the paper. Section 4 discusses the incremental anytime algorithm for interactive MOQO in detail. Section 5 proves correctness of the algorithm, analyzes its space complexity, the time complexity of a single invocation, and the amortized time of several invocations. Section 6 contains experimental results for TPC-H queries; the presented algorithm was implemented on top of the Postgres optimizer.

2. RELATED WORK

We discuss prior work solving similar problems as we do (MOQO) and prior work using similar algorithmic techniques as we do (work on anytime algorithms, incremental algorithms, and approximation algorithms). Classical query optimization [12] judges query plans based on only one cost metric. Single-objective query optimization algorithms are not applicable to MOQO in the general case; a detailed explanation can be found in prior work [7, 14]. MOQO algorithms were proposed in the context of the Mariposa system [13] where query plans are evaluated based on the two cost metrics execution time and execution fees: Papadimitriou and Yannakakis propose a fully polynomial-time approximation scheme for MOQO [11]. Their algorithm combines polynomial run time with formal approximation guarantees but does not optimize join order (only the mapping from query plan nodes to processing sites is optimized for a fixed join order) and is therefore not applicable to the query optimization problem addressed in this paper. It has been shown that even single-objective query optimization cannot be approximated in polynomial time if join order is optimized [5]; those results apply to MOQO as well since MOQO is a generalization of single-objective query optimization. Ganguly et al. [7] described an algorithm for MOQO based on dynamic programming; this algorithm produces the full set of Pareto-optimal cost tradeoffs but its execution time can be excessive in practice [14]. In prior work [14], we proposed several approximation schemes for MOQO. We assumed that users specify a preference function in the form of weights and cost bounds prior to optimization; the optimizer searches for a plan maximizing that preference function. Specifying a preference function in advance is however often difficult for users [17]; this is why we assume now that users select their preferred query plan in an interactive process. Our prior algorithms are unsuitable to be used within an interactive process since they are not incremental, meaning that consecutive invocations for the same query result in large amounts of redundant work, and since they are not anytime algorithms, meaning that they do not improve result precision in regular intervals. We discuss and justify the design constraints that an interactive interface imposes on the optimization algorithm in more detail in Section 4. During formal analysis (see Section 5) and experimental evaluation (see Section 6), we use algorithms as baseline that are very similar to the ones proposed in our prior work. Other MOQO algorithms are tailored towards specific combinations of cost metrics [18, 1]; while such special-purpose algorithms achieve good performance, they break when taking into account additional cost metrics [14]. The algorithm proposed in this paper is applicable for a broad range of plan cost metrics such as execution time, energy consumption, monetary fees, result precision and others; we cover the same metrics as the generic approximation schemes that were discussed before [14].

Anytime algorithms are algorithms whose result quality improves gradually as computation time progresses [19]; they are often applied to computationally intensive problems in situations where computation might be interrupted. MOQO is computationally intensive and user input may interrupt the current optimization at any time in our interactive scenario. Anytime algorithms have been proposed for *query processing* [16, 8] while we use them for *query optimization*. Chaudhuri motivated the development of anytime algorithms for classical query optimization [6]. We argue that anytime algorithms are even more beneficial for MOQO due to the higher computational cost and due to the additional challenge of user interaction. Incremental algorithms avoid redundant work when solving similar problem instances in consecutive invocations (e.g., when calculating shortest paths for several graphs with similar structure [2]). In our case we solve many consecutive optimization problems for the same query but with different bounds and different approximation precision. Bizarro et al. proposed an incremental algorithm for parametric query optimization [4]; plan cost depends on unknown parameters in their scenario and the optimizer might have to optimize the same query for many different combinations of parameter values. Storing result plans together with the corresponding input parameters allows to bypass future optimizer invocations for similar parameter values. Parametric query optimization is related to MOQO since both extend the problem model of classical query optimization; parametric query optimization associates plans however with cost functions while MOQO associates plans with cost vectors. MOQO algorithms are in general not applicable for parametric query optimization and vice versa, a detailed discussion of the differences can be found in prior work [15]. Existing algorithms for multi-objective parametric query optimization [15] are neither incremental nor anytime algorithms. The algorithm presented in this paper is an approximation scheme [9]: it differs from an exhaustive algorithm since it does not guarantee to return the optimal result. It offers however formal worst-case guarantees on how far the quality of the produced result is from the optimum; this distinguishes our algorithm from pure heuristics. Approximation schemes for MOQO have been proposed before [14] but they are neither anytime algorithms nor incremental. Our algorithm is iterative which connects it to other iterative query optimization algorithms [10, 14]. The algorithm proposed by Kossmann and Stocker [10] is however only applicable to single-objective query optimization while our prior iterative algorithm [14] is non-incremental, meaning that results generated in prior iterations are not reused, and the goal was to minimize total query optimization time rather than the time between consecutive results.

3. MODEL

We model a *Query* as a set Q of tables that need to be joined. We use this simple query model to describe our algorithm in Section 4 but we outline in Section 4.3 how the algorithm can be extended to support a richer query model. A *Query Plan* either scans a single table or is composed out of two *Sub-Plans* such that the result of those sub-plans is finally joined. We denote by $p = p_1 \bowtie p_2$ the plan p that uses p_1 and p_2 as sub-plans and joins their results.

Query plans are associated with scalar cost values in classical query optimization [12]. As we consider multiple cost metrics in MOQO, each plan is associated with a *Cost Vector* instead of a cost value. We denote by $\mathbf{c}(p) \in \mathbb{R}_+^l$ the cost vector associated with plan p. Each component of that vector represents the cost value according to one of the l metrics. Note that cost values are always non-negative. We use boldface for vectors (e.g., $\mathbf{c}$) to distinguish them from scalar values. The algorithm presented in Section 4 supports the same class of cost metrics as prior generic approximation schemes [14]; this set includes for instance execution time, monetary execution fees, result precision, energy consumption, or various measures of resource consumption concerning system resources such as buffer space, number of cores, or IO bandwidth. The class of supported cost metrics is characterized more thoroughly during formal analysis in Section 5. The focus of this paper is on optimization and not on costing; we do not provide our own cost formulas but assume that cost models from prior work are used to estimate the cost of query plans.

Considering one cost metric, a query plan p_1 is at least as good as another query plan p_2 if the cost of p_1 is lower than or equivalent to the cost of p_2. With multiple cost metrics, a plan p_1 is at least as good as p_2 if its cost is lower than or equivalent to the cost of p_2 according to each cost metric; if this is the case then we say that p_1 *Dominates* p_2 and denote it by $\mathbf{c}(p_1) \preceq \mathbf{c}(p_2)$. If p_1 dominates p_2 and p_1 has lower cost than p_2 according to at least one metric then we say that p_1 *Strictly Dominates* p_2 and denote it by $\mathbf{c}(p_1) \prec \mathbf{c}(p_2)$. Consider the set P of all possible plans for a fixed query: we call each plan $p^* \in P$ *Pareto-Optimal* if there is no alternative plan $p \in P$ such that $\mathbf{c}(p) \prec \mathbf{c}(p^*)$. We call the set $P^* \subseteq P$ a *Pareto Plan Set* if for each possible plan $p \in P$ there is a plan $p^* \in P^*$ with $\mathbf{c}(p^*) \preceq \mathbf{c}(p)$. Note that several subsets of P can be Pareto plan sets.

Full Pareto plan sets can be excessively large; this motivates to approximate the real Pareto set. Let $\alpha > 1$ be the *Precision Factor*. Then each subset P_α^* of P is an α-*Approximate Pareto Plan Set* if for each possible plan $p \in P$ there is a plan $p^* \in P_\alpha^*$ such that $\mathbf{c}(p^*) \preceq \alpha\mathbf{c}(p)$. Each Pareto plan set is an approximate Pareto plan set with factor $\alpha = 1$. By multiplying the cost vector of plan p by a factor greater than 1, we make its cost appear higher than it actually is; this reduces the requirements compared to the definition of the Pareto plan set. The higher α is chosen, the lower the approximation precision and the smaller the corresponding approximate Pareto plan set can be. We derive size bounds in Section 5.

Often, users care only about query plans whose cost is upper-bounded for certain cost metrics. Users might for instance have a deadline which implies an upper bound on execution time, or a monetary budget limiting execution cost. We model *Cost Bounds* by a cost vector $\mathbf{b}$ with the semantics that users are only interested in plans p with $\mathbf{c}(p) \preceq \mathbf{b}$. If $\mathbf{c}(p) \preceq \mathbf{b}$ for some plan p then we also say that it *Respects* the cost bounds while it otherwise *Exceeds* the bounds. If a user specifies cost bounds $\mathbf{b}$ then he is interested in an approximation of a subset of the Pareto plan set: an α-*Approximate* $\mathbf{b}$-*Bounded Pareto Plan Set* is a subset P^* of the set P of possible plans such that for each plan $p \in P$ with $\alpha\mathbf{c}(p) \preceq \mathbf{b}$ there is a plan $p^* \in P^*$ such that $\mathbf{c}(p^*) \preceq \alpha\mathbf{c}(p)$. The input to the *Approximate Bounded MOQO Problem* is a query Q, an approximation factor α, and cost bounds $\mathbf{b}$. The result is a α-approximate $\mathbf{b}$-bounded Pareto plan set for query Q. During *Interactive MOQO*, many approximate bounded MOQO problems may have to be solved, reflecting gradually refined approximation precision and bounds that may change due to user input.

We finally discuss the parameters used in our formal analysis: n is the number of query tables, m the cardinality of the biggest table in the data base. Parameter l is the number of cost metrics. We treat n as variable while we treat m and l as constants during complexity analysis. Those assumptions are consistent with prior work on MOQO approximation algorithms [14].

4. DESCRIPTION OF ALGORITHM

We describe the Incremental Anytime MOQO Algorithm, short IAMA, for interactive MOQO in this section. Existing MOQO algorithms [14] assume that users specify a preference function prior to optimization; the goal of optimization is to find a plan that optimizes this preference function. IAMA differs since users select the optimal query plan for their query in an interactive process. IAMA consists of two main parts: the *main control loop* and the *incremental optimizer*. The main control loop handles the interaction with the user and decides which part of the plan search space to explore next. It uses the incremental optimizer as a subfunction to generate fresh query plans. The incremental optimizer generates query plans for the given query; it allows to focus plan generation by specifying an area of interest within the plan cost space and to choose the resolution with which Pareto-optimal cost tradeoffs are approximated. Choosing a higher resolution yields more accurate results while choosing a lower resolution reduces optimization time. The main control loop uses the optimizer to increase approximation precision step by step for a given area in the cost space which leads to the anytime behavior of IAMA.

The incremental optimizer was designed with two performance constraints in mind. First, it must be *incremental*, meaning that it avoids regenerating the same query plans in consecutive invocations; this is important as the optimizer is invoked many times for a given query while only the resolution and the area of interest in the cost space change. Second, the time of any single optimizer invocation must be *proportional* to the resolution and to the size of the chosen cost space area. Both optimizer properties are crucial to enable an interactive process: If the optimizer was not incremental then each invocation would start from scratch and generating a fine-grained approximation could easily take several tens of seconds. Receiving user input during such a long time is likely which only leaves the choice between blocking the interface until optimization is finished (leading to poor user experience) or interrupting optimization without being able to reuse any results (making it unlikely that high resolutions are ever reached). If invocation time was not guaranteed to be at most proportional to the input parameters then the interface might not be able to generate a first approximation of optimal cost tradeoffs quickly.

The proposed optimizer algorithm satisfies both performance constraints. It uses a variant of the classical dynamic programming approach to query optimization [12] and generates optimal plans for joining table sets out of optimal plans for joining subsets. A single-objective optimizer would

store one cost-optimal plan per table set[1] while the IAMA optimizer might have to store many alternative query plans that all realize Pareto-optimal cost tradeoffs. The IAMA optimizer becomes incremental by maintaining two plan sets across invocations: the *result plan set* and the *candidate plan set*. Both sets may contain completed query plans, joining all tables in the current query, as well as partial query plans, joining only a subset of tables. Result plans have already been verified to be crucial in order to approximate a specific cost space area with a specific resolution. Candidate plans have only been determined to be potentially useful for a given cost space area and resolution. A future optimizer invocation will decide whether they are relevant indeed.

Both plan sets are indexed by plan cost and by resolution level. Using a data structure supporting multi-dimensional range queries allows to efficiently retrieve plans whose cost is within a certain range and which are registered for a certain range of resolution levels. Indexing plans by their cost vectors enables the optimizer to focus on certain cost space areas. Indexing candidate plans by resolution allows to avoid checking relevance of the same candidate for the same resolution twice. We will formally prove in Section 5 that the proposed algorithm indeed verifies relevance only once per resolution and candidate plan. Indexing result plans by resolution is required to guarantee that optimization time is always proportional to the chosen resolution. When inserting new partial candidate plans during an optimizer invocation, they should for instance only be combined with result plans that are registered for the current resolution level or lower. We illustrate how our algorithm works by means of a highly simplified example before providing details.

EXAMPLE 3. *We consider the two cost metrics execution time and monetary fees (e.g., in a cloud scenario) and optimize the simple query $R \bowtie S$. The user selects very tight cost bounds on execution fees. The initial goal of the optimizer is to quickly produce a coarse-grained approximation of the optimal cost tradeoffs for the query. Therefore, optimization starts with the lowest possible resolution level 0.*

The optimizer starts by considering alternatives scan plans for the two single relations R and S. If the optimizer encounters a scan plan whose cost exceed the user bounds then this plan is stored as candidate for later optimizer invocations as it might become useful once the user changes the bounds. If the optimizer encounters several plans for the same table whose cost are roughly comparable then only one of them is stored as result plan while the others are stored as candidates; the other plans might become useful once the resolution is refined. In a second step, the optimizer combines result scan plans to form join plans answering the entire query. The optimizer separates result plans from candidate plans in the same fashion as before and shows the cost of the result plans to the user.

Without user intervention, the resolution is increased to 1. Now the optimizer reconsiders some of the scan plans for R and S that were stored as candidates. The optimizer does not reconsider candidate plans whose cost exceed the bounds since the user did not change them. The optimizer reconsiders candidate plans whose cost was roughly comparable to the cost of a result plan. Two plans whose costs were considered equivalent at resolution level 0 might not be equivalent anymore at resolution level 1; such plans are inserted as result plans. Then the optimizer uses the freshly inserted result scan plans to combine fresh plans for the entire query.

Assume the user relaxes the tight bound on monetary fees. Now the resolution is reset to 0 in order to quickly generate a rough approximation of available cost tradeoffs for the new bounds. The optimizer only reconsiders candidate scan plans whose costs exceeded the previous bounds but no candidates whose cost was considered equivalent to one of the result plans at resolution 0 or 1. Freshly inserted result plans are used to combine fresh plans for the entire query; the user view is updated.

Section 4.1 describes the main control loop and Section 4.2 discusses the pseudo-code of the incremental optimizer. Section 4.3 proposes finally several extensions.

4.1 Main Control Loop

Algorithm 1 is the main function of IAMA. Its input is a query and its output is the query plan that the user selects for execution in an interactive process. Algorithm 1 contains the main control loop from lines 12 to 25; each iteration of the main loop generates new query plans by invoking the OPTIMIZE procedure (its implementation is discussed in the next subsection), visualizes their cost using the VISUALIZE procedure (we do not provide pseudo-code for this procedure), and selects the focus for the next optimizer invocation, taking into account user input, if any.

The optimization focus is described by the two local variables $\mathbf{b}$ and r. Variable $\mathbf{b}$ is a vector of upper cost bounds restricting the area of interest in the cost space. Variable $\mathbf{b}$ is used as parameter for the optimizer invocation and the optimizer focuses on generating plans that respect the cost bounds (i.e., plans whose cost vector is dominated by $\mathbf{b}$). The cost bounds are initialized to default values (this can also be the value ∞, indicating that no bounds are set by default) and can be adapted by the user in each iteration of the main control loop. Adapting the bounds gives users the opportunity to focus plan search on the relevant part of the cost space, thereby speeding up optimization. Variable r represents the resolution with which the Pareto-optimal cost tradeoffs are approximated. At a low resolution, the optimizer does not distinguish query plans with similar cost vectors and generates a relatively small set of representative query plans. At a high resolution, the optimizer generates more query plans and the approximation of the set of Pareto-optimal cost tradeoffs is therefore more fine-grained. Assuming a two-dimensional visualization of cost vectors, a high resolution translates into pixels representing alternative cost tradeoffs being closer together while a low resolution means that those pixels are far apart from each other (see Figure 1 from Section 1 for an example: the resolution increases from Figure 1(a) to Figure 1(b)). This motivates the use of the term *resolution*. We assume that a predetermined number of resolution levels is used; the value domain of variable r is the set of resolution levels $\{0, \ldots, r_M\}$. Variable r is initialized with the lowest possible resolution and is increased by one in each iteration of the main control loop if no user input is received. If the user changes the cost bounds then the resolution is set to zero again. Gradually increasing resolution allows to keep each optimizer invocation short (note that this reasoning is only valid since the

[1]Single-objective optimizers might still store several cost-optimal plans for a table set if they produce different tuple orderings that might speed up following operations; we neglect tuple orderings here to simplify the explanations.

```
1: // Generate Pareto plan set of increasing resolution
2: // for query Q until user selects query plan
3: function IncAnyMOQO(Q)
4:     // Initialize bounds and resolution
5:     b ← default bounds
6:     r ← 0
7:     // Fill in scan plans for single tables
8:     for q ∈ Q, p ∈ScanPlans(q) do
9:         Prune(Res^q, Cand^q, b, r, p)
10:    end for
11:    // Main control loop
12:    repeat
13:        // Generate more plans
14:        Optimize(Q, Res, Cand, b, r)
15:        // Visualize cost of known plans
16:        Visualize(Res^Q[0..b, 0..r])
17:        // Update bounds or refine resolution
18:        if User changed bounds then
19:            b ← user-specified bounds
20:            r ← 0
21:        else
22:            // Refine resolution until r_M is reached
23:            r ← min(r_M, r + 1)
24:        end if
25:    until User selects plan p
26:    // Return result plan
27:    return p
28: end function
```

Algorithm 1: Main function for interactive query optimization: processes user input, visualizes plan cost, and invokes incremental optimization procedure.

```
1: // Generate plans for query Q, insert them into result
2: // set Res if they are relevant for current resolution r
3: // and bounds b or insert them into candidate set Cand
4: // if they might become relevant later.
5: procedure Optimize(Q, Res, Cand, b, r)
6:     // Check candidate plans
7:     for q ⊆ Q do
8:         for p_C ∈ Cand^q[0..b, 0..r] do
9:             Cand^q ← Cand^q \ {p_C}
10:            Prune(Res^q, Cand^q, b, r, p_C)
11:        end for
12:    end for
13:    // Generate plans using fresh candidates
14:    for k ← 2 to |Q| do
15:        for q ⊆ Q : |q| = k do
16:            for q_1 ⊂ Q : q_1 ≠ ∅; q_2 ← Q \ q_1 do
17:                for p_F ∈Fresh(Res^{q_1}, Res^{q_2}, b, r) do
18:                    Prune(Res^q, Cand^q, b, r, p_F)
19:                end for
20:            end for
21:        end for
22:    end for
23: end procedure
```

Algorithm 2: Incremental optimization algorithm for multi-objective query optimization.

optimizer is incremental). Under the reasonable assumption that the time for one iteration of the main loop is mainly determined by optimization time, the short optimization times lead to high iteration frequencies. This guarantees that the plan cost visualization is updated frequently and that the interface remains responsive. Resetting the resolution after a bounds change makes sure that first results become visible quickly after the user adapts the cost space area of interest.

Variable *Res* stores the set of result plans and variable *Cand* the set of candidate plans. Both sets contain partial plans that join subsets of Q in addition to completed plans that join all tables in Q; we use the superscript notation to refer to subsets of plans that join specific table sets (e.g., Res^q for $q \subseteq Q$ denotes the subset of result plans that join table set q). Plans in both sets are also indexed by their cost vectors and by the resolution at which they were inserted (result set) or at which they should be considered for insertion (candidate set). We refer to subsets of plans that are associated with a specific resolution range and cost range using square brackets: $Res^q[\mathbf{0}..\mathbf{b}, 0..r]$ selects for instance all result plans that join table set q, were inserted at a resolution between 0 and r (both limits inclusive), and whose cost is dominated by $\mathbf{b}$. The analogous notation applies for candidate plans. Those plan selections correspond to range queries in the space that is spanned by all of the plan cost metrics and by the resolution level as additional dimension. A classic survey on data structures supporting range queries was compiled by Bentley and Friedman [3]. Different data structures offer different tradeoffs between insertion and retrieval time. We will later prove and exploit the fact that the number of plan insertions is bounded for a fixed query while the number of retrieval operations is not (see Section 5.4). Prioritizing fast retrieval over fast insertion times and selecting a corresponding data structure seems therefore advantageous.

Both sets, result plans and candidate plans, are initially empty in each invocation of Algorithm 1. They are initialized before the main control loop starts, by inserting plans for scanning single query tables using procedure Prune. The pruning procedure decides whether to insert plans into the result or candidate set and its implementation will be discussed in the next subsection. New plans can get generated and inserted into *Res* and *Cand* in each invocation of the Optimize procedure. Note that we assume call-by-reference parameter passing such that the optimizer subfunction can alter the state of *Res* and *Cand*. Procedure Visualize visualizes only cost tradeoffs of completed query plans which respect the current cost bounds and are appropriate for the current resolution; the input set to procedure Visualize is therefore the subset of completed query plans described by $Res^Q[\mathbf{0}..\mathbf{b}, 0..r]$.

4.2 Incremental Optimizer

Algorithm 2 is the incremental optimizer procedure that is invoked in each iteration of the main loop (lines 12 to 25 in Algorithm 1). It obtains as input the current query Q, the set of result and candidate plans (which it may alter), as well as cost bounds $\mathbf{b}$ and resolution r. After the optimizer invocation, the result set is guaranteed to contain a $\mathbf{b}$-bounded approximation of the Pareto plan set for query Q with resolution r. This may or may not require the optimizer to insert new plans into the result set. As the optimizer is incremental, it will only insert new plans in addition to the ones already contained in *Res* if this is required to satisfy the previously mentioned guarantee. The optimizer may also insert plans into the candidate plan set *Cand*, discard

plans from the candidate set, or re-index candidate plans for a different resolution. The purpose of the candidate set is to avoid redundant work over different optimizer invocations: a non-incremental MOQO algorithm [14] discards query plans that are not useful for the current invocation. IAMA keeps them as candidate plans instead and does not need to regenerate them in later invocations. Re-indexing and discarding candidate plans also minimizes redundant work: if it has been verified during the current optimizer invocation that a certain candidate plan is irrelevant for a given resolution then it is not necessary to recheck that candidate plan for the same resolution again in future invocations. Re-indexing that candidate plan for a higher resolution makes sure that the knowledge gained in the current invocation (about the irrelevance of that candidate) is not lost. If candidate plans are irrelevant even for the highest resolution then they can be safely discarded.

Algorithm 2 consists of two phases. In the first phase (lines 6 to 12), the optimizer *reconsiders candidate plans that were generated in previous invocations*. It iterates over all table subsets of Q in arbitrary order and retrieves for each set all candidate plans that are indexed for the current resolution and the current cost bounds. All considered plans are deleted from the candidate set and pruned; the pruning procedure might insert them again as candidates but for a higher resolution than the current one. The pruning procedure (whose pseudo-code is discussed later) might also insert them into the result plan set. The second phase of Algorithm 2 (lines 13 to 22) generates new plans by combining plans in the result sets. During that phase, the optimizer iterates over table sets of increasing cardinality; for each table set, it considers all possible splits into two non-empty subsets. For each split of a set q into two subsets q_1 and q_2, the optimizer considers combining a plan joining the tables in q_1 with one joining the tables in q_2 to obtain a plan joining all tables in q. In contrast to classical query optimization algorithms [12], the incremental optimizer does not combine all plans in the result plan sets but only considers fresh combinations of sub-plans that were not generated in prior optimizer invocations. Function FRESH (whose pseudo-code is discussed next) returns only such plans.

Algorithm 3 shows pseudo-code for the pruning procedure PRUNE and for function FRESH generating fresh query plans. Procedure PRUNE inserts a new query plan into the result set if its cost vector cannot be approximated by any alternative result plan at the current resolution. We use the expression $\text{INSERT}(S, p)$ for some set S and a plan p as shortcut for $S \leftarrow S \cup \{p\}$. Resolution levels r translate into an approximation factor α_r by which the cost vector of the new plan is multiplied before it is compared with the cost vectors of the alternative plans (line 7). The approximation factors α_r are chosen such that $\alpha_r > 1$ and $\alpha_r > \alpha_{r+1}$ for all resolution levels r; we demonstrate the effects of different choices for the number of resolution levels and approximation factors in Section 6. Scaling the cost vector of the new plan by a factor greater than one makes it more likely that the scaled vector is dominated by the cost vector of one of the alternative result plans, meaning that the new plan is not inserted into the result set; the new plan can only be inserted if its cost is for each cost metric lower than the cost of any other result plan by factor α_r at least. The higher the factor α_r, the less likely it is that the new plan is inserted. This means that the result plan set tends to grow with shrinking approximation factor and growing resolution; as the time complexity grows in the size of the result set, the complexity grows with increasing resolution as well. We calculate precise bounds in Section 5. We also show in Section 5 that an invocation of the optimizer function with resolution r yields an α_r^n-approximate Pareto plan set, where $n = |Q|$ designates the number of tables in the query. The underlying reason is intuitively that each pruning operation may in the worst case introduce an approximation error that accumulates over different pruning operations; the number of pruning operations for a single query plan is proportional to n. Knowing the relationship between α_r and the result precision allows to choose the factors α_r such that a desired target precision is still reached for the maximal expected number of tables; alternatively, the choice of α_r can be adapted to the current query. If the scaled cost vector of the new plan is dominated at the current resolution, it is inserted as candidate plan for higher resolutions or discarded if the maximal resolution has been reached. If the cost vector of the new plan exceeds the current bounds, it may become useful again once

```
1: // Insert plan p for query q into result set Res if p is
2: // relevant for current resolution r and bounds b.
3: // Insert p into candidate set Cand if p could
4: // become relevant later.
5: procedure PRUNE(Res^q, Cand^q, b, r, p)
6:     // Compare p with alternative plans and bounds
7:     if ∃p_A ∈ Res^q[0..b, 0..r] : c(p_A) ⪯ α_r · c(p) then
8:         // p_A approximates p for resolution r
9:         // → keep p as candidate for higher resolutions
10:        if r < r_M then
11:            INSERT(Cand^q[c(p), r + 1], p)
12:        end if
13:    else if c(p) ⋠ b then
14:        // Cost of p exceeds the bounds
15:        // → keep p as candidate for different bounds
16:        INSERT(Cand^q[c(p), r], p)
17:    else
18:        // p is immediately relevant
19:        // → add p to result plan set
20:        INSERT(Res^q[c(p), r], p)
21:    end if
22: end procedure
23: // Given two sets of sub-plans Res^{q1} and Res^{q2}, filter
24: // to relevant plans for resolution r and bounds b and
25: // generate all fresh combinations of sub-plans.
26: function FRESH(Res^{q1}, Res^{q2}, b, r)
27:    // Filter to relevant sub-plans
28:    P_1 ← Res^{q1}[0..b, 0..r]
29:    P_2 ← Res^{q2}[0..b, 0..r]
30:    // Generate relevant sub-plan pairs
31:    pairs ← ΔP_1 × (P_2 \ ΔP_2)
32:    pairs ← pairs ∪ ((P_1 \ ΔP_1) × ΔP_2)
33:    pairs ← pairs ∪ (ΔP_1 × ΔP_2)
34:    // Generate fresh plans
35:    fresh ← ∅
36:    for ⟨p_1, p_2⟩ ∈ pairs :ISFRESH(p_1, p_2) do
37:        fresh ← fresh ∪ {p_1 ⋈ p_2}
38:    end for
39:    return fresh
40: end function
```

Algorithm 3: Sub-functions of the optimization procedure.

the bounds change; in that case the new plan is therefore inserted as candidate for the current resolution again.

As discussed before, our goal is to make the time complexity of one optimizer invocation proportional to the current resolution and cost bounds, independently from how many candidate and result plans have accumulated from prior invocations. This goal leads to two subtle design decisions concerning the pruning function that are nevertheless crucial in order to obtain the complexity properties we were aiming for: First, the new plan is only compared to alternative plans that have been inserted at the current resolution level or at a lower one. The disadvantage is that we might insert the new plan even if plans that are preferable over the new plan were already inserted at a higher resolution; the advantage is however that the number of plan comparisons is proportional to the size of the result plan set at the current resolution. The second decision is that we do not discard result plans that are dominated by the new plan in case that the new plan is inserted into the result set. This differs from prior MOQO approximation schemes which always keep the result plan sets as small as possible [14]. The reason that we do not discard result plans is that they might have been used already as sub-plans to combine other query plans in prior invocations of the optimizer; this might have happened at the current resolution or at a higher one. Discarding a result plan would require to discard at the same time all plans that use it as sub-plan to keep the result plan sets for different table sets consistent (we also assume that plans are represented by pointers to their sub-plans as discussed in Section 5.2); the number of plans to discard is not necessarily proportional to the size of the result plan set at the current resolution. We renounce discarding result plans to keep the time complexity of the current optimizer invocation proportional to the current resolution.

Function FRESH uses result plans for two table subsets q_1 and q_2 to combine new plans that are *fresh*, i.e., they have not been generated in prior optimizer invocations. The expression ΔS designates for some plan set S a subset of plans that potentially were not yet combined with all other plans indexed for the current resolution and cost range. During invocation series in which the bounds are tightened while resolution is refined, we can include all plans that were inserted in the current invocation in ΔS (in such cases we are sure that all previously inserted plans respecting the current bounds were already combined with each other) and set $\Delta S = S$ otherwise. We can use auxiliary data structures that index plans based on the invocation at which they were inserted in combination with the index on cost and resolution; this allows to evaluate the expressions ΔS and $(S \backslash \Delta S)$ efficiently. For each cross product between plan sets, we check first if one of the two operand sets is empty before evaluating the entire expression. Predicate ISFRESH evaluates to true for plans that were not yet combined in prior invocations; we can use a hash table to perform this check efficiently. Fresh plans are returned and will be pruned.

4.3 Extensions

The algorithm presented in the last subsections is simplified and does not possess several features that are standard in query optimization. The code can easily be extended to incorporate the features discussed next and the implementation used for our experiments in Section 6 supports them as well. First, the presented code only optimizes join order but not the choice of join operators. Supporting different join operators just requires to add an inner loop iterating over all applicable join operators and creating corresponding plans in function FRESH (see Algorithm 3). Second, alternative operators might produce the same set of result tuples while some of them generate them in an order that can be exploited by future operations. This is why dynamic-programming based query optimizers distinguish plans generating different *interesting tuple orders* [12] during pruning; the cost-based plan comparison is restricted to plans generating similar tuple orders and it is straight-forward to generalize this principle to the multi-objective case. Third, the presented code is based on a simple query model, representing queries as sets of tables that need to be joined. Predicates and projections can be handled by applying them as early as possible in the join tree and the required code extensions are standard [12]. The seminal paper by Selinger [12] describes how complex SQL statements containing nested queries can be decomposed into simple select-project-join query blocks that can be optimized by our algorithm.

5. FORMAL ANALYSIS

We analyze the algorithm presented in Section 4: More precisely, we analyze the optimizer sub-function that is represented in Algorithm 2. Section 5.1 proves formal worst-case guarantees on how closely the result plan sets, produced by the optimizer, approximate the real Pareto plan set. Section 5.2 analyzes space complexity and Section 5.3 analyzes the time complexity of a single optimizer invocation. In Section 5.4, we analyze the amortized time complexity of several consecutive optimizer invocations for the same query.

5.1 Result Precision

The following analysis is based on the *Principle of Near-Optimality* [14] (PONO) for MOQO which states that replacing optimal sub-plans within a complete query plan by near-optimal sub-plans still yields a near-optimal complete plan for a broad class of cost metrics. The class of cost metrics to which the PONO applies is characterized by the *Aggregation Function*, i.e. by the recursive function that calculates the cost of a plan according to that metric out of the cost of the two sub-plans: the PONO applies to all cost metrics whose aggregation function can be represented using a combination of the operators sum, maximum, minimum, and multiplication by a constant. This applies for instance to metrics such as energy consumption or execution time[2]. The PONO has also been shown to apply for several other metrics whose aggregation formulas do not fit into the latter scheme, such as failure resilience or result precision. A formal definition of the PONO follows.

DEFINITION 1 (PONO). *Let p be a query plan with sub-plans p_1 and p_2 and pick an arbitrary $\alpha \geq 1$. Derive p^* from p by replacing p_1 by p_1^* and p_2 by p_2^*. Then $\mathbf{c}(p_1^*) \preceq \alpha \mathbf{c}(p_1)$ and $\mathbf{c}(p_2^*) \preceq \alpha \mathbf{c}(p_2)$ together imply $\mathbf{c}(p^*) \preceq \alpha \mathbf{c}(p)$.*

The following theorems are based on the PONO. We also assume *Monotone Cost Aggregation*, meaning that the cost of a plan must be higher or equal to the cost of its sub-plans according to each cost metric.

[2]The energy consumption of a plan is the sum of the energy consumption of the sub-plans. The plan execution time is the maximum of the execution times of the sub-plans for parallel execution, and the sum for sequential execution.

THEOREM 1. *After invoking* OPTIMIZE *with bounds* $\mathbf{b}$ *and resolution* r *for query* Q, $Res^q[\mathbf{0}..\mathbf{b}, 0..r]$ *contains an* α_r*-approximate* $\mathbf{b}$*-bounded Pareto plan set for each table* $q \in Q$.

PROOF. For each table q, all applicable scan plans are generated and pruned before the main loop starts. Let p be an arbitrary scan plan for an arbitrary table q. Once procedure OPTIMIZE is invoked later for resolution r and bounds $\mathbf{b}$, there are two possibilities for p: either p was inserted into the result plan set in prior invocations or it is not in the result plan set at the start of the current invocation. If p was not inserted before then we must make sure that it is either inserted in the current invocation or not required to form an α_r-approximate $\mathbf{b}$-bounded Pareto plan set.

If p was not inserted before then it must be included in $Cand^q[\mathbf{0}..\mathbf{b}, 0..r]$ unless it exceeds the bounds $\mathbf{b}$ or can be approximated by an alternative plan. In both cases, p is not required for an α_r-approximate $\mathbf{b}$-bounded Pareto plan set. If p is however in $Cand^q[\mathbf{0}..\mathbf{b}, 0..r]$ at the start of the current invocation then procedure OPTIMIZE will retrieve and prune p; plan p will be inserted if it is required for an α_r-approximate $\mathbf{b}$-bounded Pareto plan set. □

THEOREM 2. *After invoking* OPTIMIZE *with bounds* $\mathbf{b}$ *and resolution* r *for query* Q, $Res^q[\mathbf{0}..\mathbf{b}, 0..r]$ *contains an* α_r^k*-approximate* $\mathbf{b}$*-bounded Pareto plan set for each table subset* $q \subseteq Q$ *with cardinality* $k = |q|$.

PROOF. The proof is an induction over the number of tables k. Theorem 1 proves the induction start for $k = 1$. Assume that the inductional assumption has been proven for all $k < K$. Let $q \subseteq Q$ be a set of K tables and p an arbitrary plan that joins those tables with $\alpha_r^K \mathbf{c}(p) \preceq \mathbf{b}$. Plan p must have two sub-plans p_1 and p_2 that each join at most $K-1$ tables. Let q_1 and q_2 be the set of tables joined by p_1 and p_2 respectively. We assume monotone cost aggregation which implies $\alpha_r^K \mathbf{c}(p_1) \preceq \mathbf{b}$ and $\alpha_r^K \mathbf{c}(p_2) \preceq \mathbf{b}$. The inductional assumption applies to p_1 and p_2 such that $Res^{q_1}[\mathbf{0}..\mathbf{b}, 0..r]$ will contain a plan p_1^* with $\mathbf{c}(p_1^*) \preceq \alpha_r^{K-1}\mathbf{c}(p_1)$ and $Res^{q_2}[\mathbf{0}..\mathbf{b}, 0..r]$ will contain a plan p_2^* with $\mathbf{c}(p_2^*) \preceq \alpha_r^{K-1}\mathbf{c}(p_2)$ after the optimizer invocation. Plans p_1^* and p_2^* can be combined into a plan p^* that joins the same tables as p and has similar cost according to the PONO: $\mathbf{c}(p^*) \preceq \alpha_r^{K-1}\mathbf{c}(p)$. Plan p^* is generated either in the current optimizer invocation with resolution r and bounds $\mathbf{b}$ or in one of the prior invocations. If p^* is generated in the current invocation then it is inserted unless an alternative plan p^{**} with $\mathbf{c}(p^{**}) \preceq \alpha_r \mathbf{c}(p^*) \preceq \alpha_r^K \mathbf{c}(p)$ is already in the result set. In that case the theorem holds. If p^* was generated in prior invocations then it was either inserted into the result set, or it was already pruned at resolution r and its cost too similar to one of the result plans, or it will be pruned in the current iteration. In all cases the theorem holds. □

Knowing the relationship between the precision factors α_r and the approximation quality of the result plan sets allows to choose the factor α_{r_M} for the maximal resolution in function of the desired target precision.

5.2 Space Consumption

The optimizer (meaning procedure OPTIMIZE, see Algorithm 2) is called once per iteration of the main loop. We analyze the accumulated space consumption of several optimizer invocations for the same query. We denote the resolution used in the i-th invocation by r_i and the cost bounds used in the i-th invocation by $\mathbf{b}_i$. Resolution $r = \min_i r_i$ designates the minimal resolution used over all invocations and vector $\mathbf{b}$ dominates all used cost bounds: $\forall i : \mathbf{b}_i \preceq \mathbf{b}$. Result and candidate plan sets are the variables with dominant space consumption in IAMA. We upper-bound the number of plans stored in those sets after all optimizer invocations to obtain the accumulated space complexity.

LEMMA 1. *Let* q *be a set containing* k *tables. Then* Res^q *contains* $O(k^l \log_{\alpha_r}^l(m))$ *result plans.*

PROOF. The cost of a query plan joining k tables is asymptotically bounded by $O(m^{2k})$ for a broad range of cost metrics [14]. Given an approximation factor $\alpha_r > 1$, the number of cost values in the interval $[1, m^{2k}]$ such that there are no two cost values c_1 and c_2 with $c_1 \leq \alpha_r c_2$ and $c_2 \leq \alpha_r c_1$ is bounded by $O(k \log_{\alpha_r}(m))$. Generalizing to l dimensions, the number of cost vectors taken from $[1, m^{2k}]^l$ such that there are no two vectors $\mathbf{c}_1$ and $\mathbf{c}_2$ with $\mathbf{c}_1 \preceq \alpha_r \mathbf{c}_2$ and $\mathbf{c}_2 \preceq \alpha_r \mathbf{c}_1$ is bounded by $O(k^l \log_{\alpha_r}^l(m))$. The pruning function of IAMA only inserts query plans into the result set if their cost vectors cannot be approximated by any other plan in the result set, using approximation factor α_r for the comparison. Therefore, the result set for q can never contain two plans whose cost vectors can mutually approximate each other. So the bound on the number of cost vectors translates into a bound on the number of plans. □

The preceding lemma exploits an upper bound on the plan cost derived from the number of joined tables. The cost bounds additionally restrict the maximal number of result plans since plans are only inserted into the result set if they respect the current bounds. We denote by $rpt(k, \mathbf{b}, r)$ the asymptotic upper bound on the number of result plans joining a set of k tables when using bounds $\mathbf{b}$ and resolution r. The next lemma derives a bound on the number of candidate plans from the bound on the number of result plans.

LEMMA 2. $Cand^q$ *contains* $O(2^k rpt^2(k, \mathbf{b}, r))$ *candidate plans for a table set* q *with* k *tables.*

PROOF. Each candidate plan for q is constructed by combining two result plans that join subsets of q. There are $O(2^k)$ possibilities of splitting a set with k tables into two subsets. Assume that q is split into two subsets q_1 and q_2. The cardinalities of Res^{q_1} and Res^{q_2} are both bounded by $rpt(k, \mathbf{b}, r)$ since rpt grows monotonically in k and $|q_1|, |q_2| < k$. The number of possible splits times the number of subplan combinations bounds the number of candidates. □

We refer to the asymptotic upper bound on the number of candidate plans per table set by $cpt(k, \mathbf{b}, r)$ in the following. The total space complexity of IAMA is obtained by summing the number of result and candidate plans over all table sets.

THEOREM 3. *The accumulated space consumption of several optimizer invocations for an* n*-table query is in* $O(3^n rpt^2(n, \mathbf{b}, r))$.

PROOF. Each plan can be represented in $O(1)$ space: scan plans are represented by the ID of the table being scanned; other plans can be represented by the IDs of the two subplans generating the operands for the final join. Plan cost vectors have constant space consumption since l is treated as a constant (see Section 3). We assume $O(l) = O(1)$ indexing

space overhead per plan which is true for many data structures supporting range queries, including the cell data structure [3]. The number of candidate plans dominates the number of result plans. Summing over all table sets we obtain a space complexity of $O(\sum_{k=1}^{n} \binom{n}{k} cpt(k, \mathbf{b}, r))$. Using the definition of cpt, considering that $cpt(k, \mathbf{b}, r) \leq cpt(n, \mathbf{b}, r)$ for $k \leq n$, and exploiting $\sum_{k=0}^{n} \binom{n}{k} 2^k = 3^n$ yields the final complexity. □

5.3 Time of Single Optimizer Invocation

We analyze the time complexity of a single optimizer invocation for a query with n tables, for bounds $\mathbf{b}$, and for resolution r. The following analysis is valid independently from which invocations of OPTIMIZE precede the analyzed invocation. We simplify and assume that retrieving F plans by a range query takes $O(F)$ time. We can for instance use a data structure similar to the cell data structure, described by Bentley and Friedmann [3]: we partition the resolution and plan cost space into cells[3], associate a list of plans with every cell, and make those lists accessible via direct lookup. Assuming suitable cell sizes and plan cost distributions such that the number of empty cells as well as the number of plans in partially included cells is negligible for most range queries, retrieval is in $O(F)$ time and single plan insertion in $O(1)$.

LEMMA 3. *Invoking* PRUNE *for a plan joining k tables is in $O(rpt(k, \mathbf{b}, r))$ time.*

PROOF. The pruning procedure retrieves all result plans joining the same tables as the new plan if they respect the bounds $\mathbf{b}$ and are indexed for resolution r or smaller. The number of plans is in $O(rpt(k, \mathbf{b}, r))$ and so is the retrieval time. The cost vector of the new plan is compared against the cost vectors of all retrieved plans. One comparison requires to compare l cost values but l is a constant (see Section 3). Adding the new plan takes constant time. □

LEMMA 4. *Invoking* FRESH *for two table sets with maximally k tables is in $O(rpt^2(k, \mathbf{b}, r))$ time.*

PROOF. Function FRESH iterates over pairs of result plans. The plans from those sets are combined pair-wise forming $O(rpt^2(k, \mathbf{b}, r))$ pairs. Constructing a new plan and calculating its cost from the cached cost of the sub-plans using recursive cost formulas is in $O(1)$. □

We use the previous results to calculate the time complexity of the OPTIMIZE procedure.

THEOREM 4. *Invoking* OPTIMIZE *for a query with n tables is in $O(3^n rpt^3(n, \mathbf{b}, r))$ time.*

PROOF. The first part of the OPTIMIZE procedure checks which candidate plans have become relevant. For one table set with k tables, this requires to retrieve and prune all candidate plans that respect bounds $\mathbf{b}$ and are marked as potentially relevant for resolution r or smaller which takes $O(cpt(k, \mathbf{b}, r) rpt(k, \mathbf{b}, r)) = O(2^k rpt^3(k, \mathbf{b}, r))$ time. The second part of the OPTIMIZE procedure generates fresh plans using the newly inserted result plans and prunes the generated new plans. For one table set with k tables, this requires again $O(2^k rpt^3(k, \mathbf{b}, r))$ time, using the complexity results for pruning and plan generation. Summing over all table sets yields a time complexity of $O(\sum_{k=1}^{n} \binom{n}{k} 2^k rpt^3(k, \mathbf{b}, r))$ which is in $O(3^n rpt^3(n, \mathbf{b}, r))$. □

[3]We can use logarithmic partitioning for the cost space which should lead to a more uniform distribution of plans over cells since the area in the cost space that a result plan approximately dominates is defined by multiplying its cost vector by a constant factor.

The time complexity of one optimizer invocation only depends on the parameters (resolution and cost bounds) of the current invocation but not on parameters used for previous invocations. This means that plans stored from previous iterations do not cause any time overhead.

5.4 Amortized Optimization Time over Several Optimizer Invocations

We analyzed the time complexity of a single optimizer invocation in the preceding subsection. Now we analyze the amortized time complexity of a large series of invocations for the same query. After each invocation, the optimizer keeps plans that could be relevant for future invocations, thereby avoiding redundant computation. The amortized time complexity of a series of invocation is therefore lower than the time complexity of a single invocation. Resolution r and bounds $\mathbf{b}$ vary between invocations while query Q is fixed. We assume an invocation series where the Δ operator in function FRESH filters plans to the ones that were inserted in the current invocation (e.g., if the user keeps tightening the bounds and the resolution is refined). The next lemmata bound the amount of redundant computation.

LEMMA 5. *Each possible plan is generated at most once.*

PROOF. Scan plans are only generated before the main loop of Algorithm 1 is entered; this code is executed only once per query. Other plans are only generated in function FRESH and we explicitly make sure to generate plans only for fresh sub-plan combinations using predicate ISFRESH. □

LEMMA 6. *Each sub-plan pair is generated at most once.*

PROOF. Each possible plan is inserted at most once into the result plan set since it is generated at most once (according to the previous lemma) and since each plan is removed from the candidate set once it is inserted into the result set. Assume that optimizer invocations are numbered and denote for two arbitrary plans p_1 and p_2 by i_1 and i_2 the invocations at which they become result plans. Then the corresponding plan pair can only be considered at invocation $\max(i_1, i_2)$: it cannot be considered before since at least one of the plans is not in the result set at this point and it cannot be considered afterwards since none of the two plans was freshly inserted at that time. □

Candidate plans are considered for insertion (into the result set) in the first phase of the OPTIMIZE procedure. Each possible plan is only considered a limited number of times according to the following lemma.

LEMMA 7. *Each generated plan is retrieved at most r_M+1 times from the candidate plan set.*

PROOF. Each retrieved candidate plan is deleted from the candidates and pruned. During pruning, the plan can be inserted as candidate again (and considered in future invocations). All considered plans respect the current bounds.

Therefore, no plan can be re-inserted as candidate because it exceeds the bounds. It can only be re-inserted as candidate if it is approximately dominated by another plan. But then the plan becomes candidate only for a higher resolution than the current one. As there are only $r_M + 1$ resolution levels, the plan can be re-considered only so many times. □

The preceding two lemmata bound the total amount of work that is necessary per query plan over several optimizer invocations. The following theorem analyzes amortized complexity of a large series of optimizer invocations.

THEOREM 5. *Procedure* OPTIMIZE *has amortized time complexity* $O(3^n)$ *for a large number of invocations.*

PROOF. We split time T^i_{opt} for the i-th optimizer invocation into a time component T^i_{dep} that depends on the number of retrieved and generated plans and another time component T^i_{idp} which does not, such that $T^i_{opt} = T^i_{dep} + T^i_{idp}$.

We express T^i_{dep} in the following. Newly generated or retrieved plans must be pruned and we assume that pruning time dominates plan generation, retrieval, and insertion time. Let s_i be the number of plans and plan pairs that were generated or retrieved as candidates in the i-th invocation. The pruning time for a query with n tables must be in $O(rpt(n, \infty, r_M))$, using Lemma 3 and the fact that pruning time becomes maximal for the highest resolution and without bounds. Hence we obtain $T^i_{dep} \in O(s_i \cdot rpt(n, \infty, r_M))$.

Even if no plans are retrieved or generated, there is still time overhead for verifying whether candidate plans have to be pruned or fresh plans can be generated. This requires us to iterate over all table sets (searching for candidates) and to iterate over each split of each table set (searching for fresh plans). Using a similar reasoning as in the previous proofs, we obtain $T^i_{idp} \in O(3^n)$.

The time $T = \sum_{i=1}^{x} T^i_{opt}$ denotes the time of x consecutive optimizer invocations for the same query. We certainly have $T \in O((rpt(n, \infty, r_M) \cdot \sum_{i=1}^{x} s_i) + x \cdot 3^n)$. However, as the total number of generated plans for a fixed query is bounded (see Section 5.2), as each plan and plan pair is generated only once (Lemmata 5 and 6), and as each plan is retrieved at most $r_M + 1$ times (Lemma 7), we can bound $\sum_{i=1}^{x} s_i$ independently from the number of invocations x. This means that for a sufficiently large number of invocations, the time component that is independent of the number of retrieved and generated plans must become dominant. □

As IAMA avoids redundant work, its averaged time complexity over many iterations equals the time complexity of single-objective query optimization with bushy plans.

6. EXPERIMENTAL EVALUATION

Section 6.1 describes and justifies the experimental setup and Section 6.2 discusses the experimental results.

6.1 Experimental Setup

We evaluate an incremental anytime algorithm for MOQO, its simplified pseudo-code was presented in Section 4, in comparison with two baselines: the *memoryless algorithm* is equivalent to the iterative MOQO algorithm proposed in a prior publication [14] except that we use a different precision refinement policy, the *one-shot algorithm* corresponds to the non-iterative MOQO algorithm presented in the same publication [14]. The memoryless algorithm produces the same sequence of result plan sets as the incremental anytime algorithm; it is however non-incremental and produces each plan set from scratch. The one-shot algorithm produces the result plan set with highest resolution directly, avoiding any intermediate steps; it therefore lacks the anytime property and takes a long time to produce the first result. We compare the algorithms according to average and maximal time of a single optimizer invocation within a series of invocations for the same query. It is crucial to minimize the time for single optimizer invocations in an interactive scenario: if single optimizer invocations take too long then it is unlikely that they won't be interrupted by user interaction. We do not evaluate space consumption: all three evaluated algorithms finally produce a result plan set with the same resolution so the total space consumption does not differ significantly between them. We evaluate all algorithms in a scenario without user interaction to make the comparison as fair as possible; the cost bounds are initially fixed to ∞.

Our implementation is based on an extended version of the Postgres 9.2 database system: this version features an optimizer that considers multiple plan cost metrics and was already used in prior work to evaluate MOQO algorithms [14]. We reuse the cost models of the three plan cost metrics execution time, consumed system resources (namely the number of reserved cores), and result precision. We chose a scenario with three plan cost metrics on purpose since this is the maximal number of metrics that allows to visualize Pareto-optimal cost tradeoffs to the user (in the form of a surface in 3D); it is of course still possible to provide users with aggregate information about available cost tradeoffs for higher numbers of metrics. Our implementation only covers the parts of the optimization algorithms that are required for this benchmark. We evaluate the algorithms on TPC-H queries containing at least one join. The performance of the algorithms is strongly correlated with the number of joined tables. In the following figures, we report average numbers over all queries with the same number of tables to make those correlations visible. Also, the Postgres optimizer may split up optimization of one TPC-H query into multiple optimizations of sub-queries with different numbers of tables. In those cases, we measured optimization times for different sub-queries separately. The relative performance of the evaluated algorithms also depends on the number of resolution levels. We experimented with different numbers of resolution levels (i.e., we used different values for r_M) and applied the formula $\alpha_r = \alpha_T + \alpha_S(r_M - r)/r_M$ to calculate the precision factors used during pruning; we use different values for the target precision α_T and for the precision step α_S. All experiments were executed on a MacBook air with 4 GB RAM and an Intel Core i5 processor with 1.4 GHz.

6.2 Experimental Results

Figure 3 shows average times per optimizer invocation for a moderate target precision of $\alpha_T = 1.01$ and $\alpha_S = 0.05$. Choosing $\alpha_T = 1.01$ means that the final result plan set is an $1.01^8 \approx 1.08$-approximate Pareto plan set, based on the formal analysis from Section 5.1 and on the fact that TPC-H queries have at most eight tables. Hence the costs of the result query plans are formally guaranteed to be not higher than optimal by more than about 8 percent. These are rather weak guarantees and, correspondingly, optimization takes never more than ten seconds, even for the two baselines. Such optimization times are not unusual for

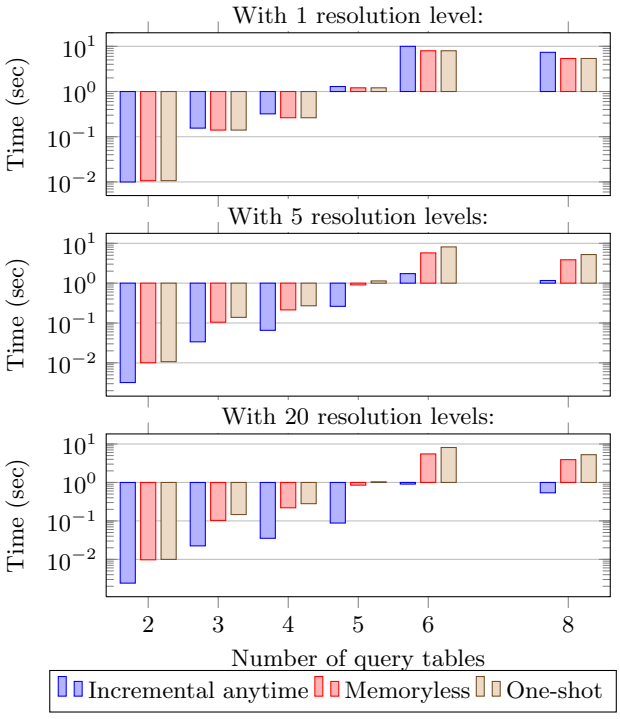


Figure 3: Average time per optimizer invocation for TPC-H sub-queries and target precision $\alpha_T = 1.01$

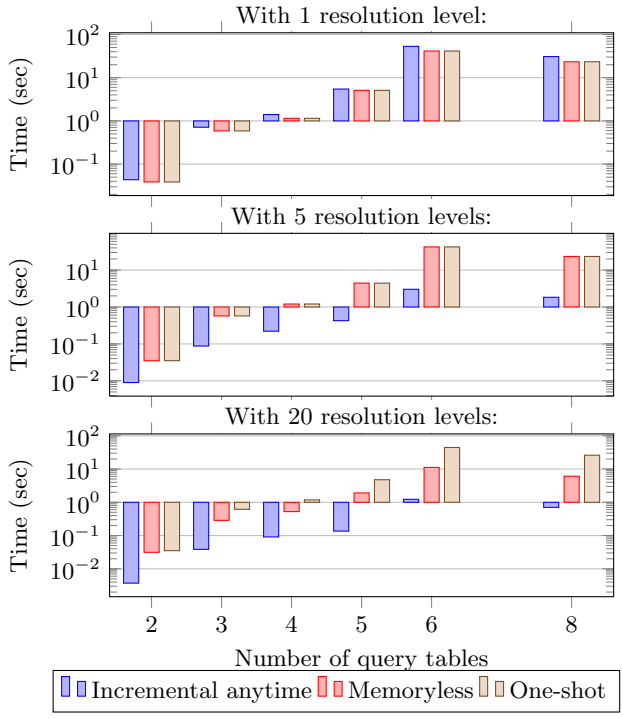


Figure 4: Average time per optimizer invocation for TPC-H sub-queries and target precision $\alpha_T = 1.005$

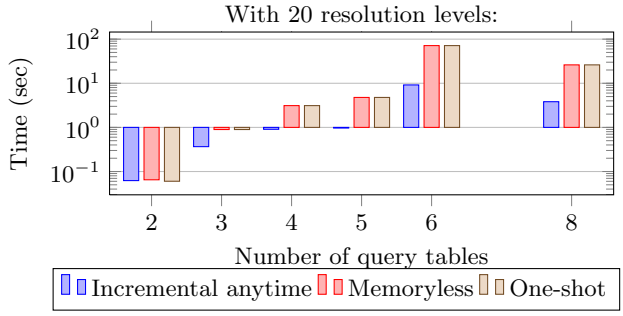


Figure 5: Maximal time per optimizer invocation for TPC-H sub-queries and target precision $\alpha_T = 1.005$

MOQO [14]. The plan search space size increases in the number of query tables and so do optimization times[4]. Note that no TPC-H sub-query joins seven tables which is the reason for the missing bar at that position. When considering only one resolution level, the incremental anytime algorithm (IAMA) cannot show its strengths and is slower than the two baselines by at most 37%. This overhead is due to plan indexing and the extended pruning function. The situation changes once we increase the number of resolution levels: already with five resolution levels, IAMA is up to four times faster than the one-shot algorithm and up to three times faster than the memoryless algorithm. With 20 resolution levels, IAMA is up to one order of magnitude faster than both baselines. Only IAMA is able to exploit different resolution levels by splitting up optimization into several incremental optimization steps. The behavior of the one-shot algorithm does not depend on the number of resolution levels; the memoryless algorithm generates the same sequence of result plan sets as IAMA but is not incremental and has to start optimization from scratch in each invocation.

Figure 4 shows analogous results for target precision $\alpha_T = 1.005$ (and $\alpha_S = 0.5$); using that target precision during pruning, all evaluated algorithms guarantee to generate 1.04-approximate Pareto plan sets so the precision is higher than before. This results in optimization times of between 41 and 53 seconds for all three algorithms with one resolution level. This makes incremental computation even more necessary than in the last example. IAMA is up to 14 times faster than the memoryless algorithm and beats the one-shot algorithm by up to factor 37. This means that the relative advantage that IAMA gives over non-incremental algorithms increases, the more difficult the optimization task is (e.g., higher target precision or higher number of tables). Figure 5 finally shows not the average but the maximal time for one optimizer invocation: IAMA is up to eight times faster than both baselines and we believe that this ratio could be extended by a more optimized sequence of precision factors. The two baselines are in practice equivalent when considering maximum time: for the memoryless algorithm, the invocation with maximal execution time is usually the last one in which it has to accomplish the same work as the one-shot algorithm.

[4]There is a slight decrease from six to eight tables since the only TPC-H query joining eight tables refers to many small tables for which less sampling strategies are considered.

7. CONCLUSION

User preferences are difficult to formalize so MOQO should be an interactive process. We presented an incremental anytime algorithm that is well suited for interactive MOQO.

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] G. Ausiello, G. F. Italiano, A. Marchetti Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–638, 1991.

[3] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

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

[5] S. Chatterji and S. Evani. On the complexity of approximate query optimization. In *PODS*, pages 282–292, 2002.

[6] S. Chaudhuri. Query optimizers: time to rethink the contract? *SIGMOD*, pages 961–968, 2009.

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

[8] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD*, 26(2):171–182, June 1997.

[9] P. Klein and N. Young. Approximation algorithms for NP-hard optimization problems. In *Algorithms and Theory of Computation Handbook*. 2010.

[10] D. Kossmann and K. Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM TODS*, 25(1):43–82, 2000.

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

[12] 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.

[13] M. Stonebraker, P. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: a new architecture for distributed data. In *ICDE*, pages 54–65, 1994.

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

[15] I. Trummer and C. Koch. Multi-objective parametric query optimization. *VLDB*, 8(3):221–232, 2015.

[16] S. V. Vrbsky and Jane W.-S. Liu. APPROXIMATE-a query processor that produces monotonically improving approximate answers. *KDE*, 5(6):1056–1068, 1993.

[17] R. Willis, CEWillis, C., & Perlack. Multiple objective decision making: generating techniques or goal programming. *Journal of the Northeastern Agr. Econ. Council*, 9(1):0–5, 1980.

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

[19] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.