# From Massive Parallelization to Quantum Computing: Seven Novel Approaches to Query Optimization

THÈSE N$^{O}$ 6995 (2016)

PRÉSENTÉE LE 16 JUIN 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE THÉORIE ET APPLICATIONS D'ANALYSE DE DONNÉES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Immanuel TRUMMER

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Study hard what interests you the most
in the most undisciplined, irreverent
and original manner possible.
— Richard Feynman

Query optimization is not rocket science.
When you flunk out of query optimization,
we make you go build rockets.
— David DeWitt

To my family and friends . . .

# Acknowledgements

First of all, I want to thank my supervisor, Christoph Koch, without whom this dissertation would not have been possible. Thanks, Christoph, for your faith in me, for your advice on research, career, and life in general, for working around the clock, and for creating just the right environment for research!

I want to thank Alon Halevy for his inspiration and advice, for an awesome internship and research experience, for being in my thesis committee, and for his continued support throughout various stages of my academic career. I equally want to thank Hongrae Lee and Sunita Sarawagi for their advice and inspiration during my stay at Google Mountain View and for their help and support during the last years. I want to thank Aditya Parameswaran, Martin Odersky, and Volkan Cevher for their helpful advice and for being part of my thesis jury. I want to thank Walter Binder for his advice and for his support at various occasions. I want to thank my other co-authors (in alphabetical order) Karl Aberer, Daniele Bonetta, Rahul Gupta, Frank Leymann, Ralph Mietzner, Diego Milano, Thanasis Papaioannou, Cesare Pautasso, Achille Peternier, Mehdi Riahi, Heiko Schuldt, Nenad Stojnic for their advice and help. I want to thank Davide Venturelli and Sergio Boixo for their support and helpful comments on my research on quantum annealing. I also want to thank Anastasia Ailamaki for some great seminars on databases. In particular, I want to thank my former supervisor, Boi Faltings, for his support and advice, for inviting me to EPFL, and for introducing me to the areas of AI and optimization.

I want to thank my long-term friends at EPFL, in particular Ayush, Adrian, Alina, Andrew, Cristina, Daniel, Eleni, Mehdi, Michele, Mirjana, Thomas, and Valentina for so many pleasant hours that have made my PhD so much more enjoyable! I also want to thank all my long-term friends in Stuttgart, in particular Andi, Barbara, David, Hanne, Helen, Ilka, Marc, Martin, Matze, Oskar, Robert, Sabrina, Salome, and Sebi for being my friends since so many years! I want to thank all my lab mates for their help and motivation and Simone for her patience and support. I want to say "Vielen Dank!" to Fabienne for her care and support during my dissertation time. And a big "ngek" goes to Marina for her care and support and for all the fun that we had together. And finally, I want to thank my family for having made me the person that I am today and thereby having laid the fundament for this dissertation.

I want to thank Google for supporting me by a European PhD fellowship and an awesome internship experience. I want to thank the Universities Space Research Association for a grant for computation time on a D-Wave adiabatic quantum annealer. And I want to thank EPFL for having provided me with an awesome environment to pursue my PhD studies.

*Lausanne, 1.1. 2016* I. T.

# Zusammenfassung

Anfrageoptimierung ist ein Schlüsselproblem im Bereich der Datenbanksysteme. Das Ziel ist es, einen Anfrage (welche die zu generierenden Daten beschreibt) in einen effizienten Plan zu überführen (welcher einen Weg beschreibt, um die angefragten Daten zu generieren). Anfrageoptimierung gehört zur Klasse der NP-harten Optimierungsprobleme und die Grösse der Probleminstanzen, die in annehmbarer Zeit gelöst werden können, ist daher in der Praxis beschränkt. In dieser Doktorarbeit stelle ich sieben neuartige Verfahren vor, mit deren Hilfe wir deutlich grössere Probleme als mit den vorher existierenden Verfahren lösen können. Diese sieben Verfahren beziehen sich auf fünf verschiedene Varianten der Anfrageoptimierung, insbesondere auf das klassische Anfrageoptimierungsproblem, auf Anfrageoptimierung mit Parametern, auf Anfrageoptimierung mit mehreren Kostenmetriken, auf Anfrageoptimierung mit Parametern und mehreren Kostenmetriken, und auf Anfrageoptimierung mit mehreren Anfragen.

Die ersten Kapitel dieser Arbeit beziehen sich vor allem auf Anfrageoptimierung mit mehreren Kostenmetriken. Mit Hilfe von Näherungsverfahren können wir näherungsweise optimale Pläne innerhalb von Sekunden finden für Probleme, bei denen es Stunden dauern würde, einen garantiert optimalen Plan zu finden. Wir stellen ausserdem einen inkrementellen Algorithmus vor, der es Benutzern ermöglicht, den präferierten Plan in einem interaktiven Prozess zu bestimmen. Desweiteren führen wir eine neue Problemvariante der Anfrageoptimierung, Anfrageoptimierung mit mehreren Parametern und Kostenmetriken, ein. Indem wir dieses Problem lösen, können wir Anfrageoptimierung vor der Laufzeit durchführen und somit Verzögerungen während der Ausführung vermeiden. Ausserdem präsentieren wir den ersten stochastischen Algorithmus für Anfrageoptimierung mit mehreren Kostenmetriken. In den späteren Kapiteln dieser Arbeit wenden wir uns anderen Varianten der Anfrageoptimierung zu. Wir stellen einen Ansatz vor, der es erlaubt zahlreiche Problemvarianten der Anfrageoptimierung innerhalb grosser Computersysteme zu parallelisieren. Anfragen werden heutzutage mit Hilfe von hoch-parallelen Systemen verarbeitet und es gibt keinen Grund, warum man dieselben Systeme nicht auch für die Optimierung verwenden sollte. Wir stellen ebenfalls einen Ansatz vor, der es erlaubt, Anfrageoptimierungsprobleme mit Hilfe linearer Programme anzunähern. Dies erlaubt es, spezialisierte Software zu verwenden, um die letztgenannten Probleme zu lösen. Dieser Ansatz erlaubt es uns, deutlich grössere Suchräume zu durchforsten als es mit traditionellen Methoden möglich ist. Schlussendlich zeigen wir, wie eine Anfrageoptimierungsvariante mit Hilfe eines Quantencomputers gelöst werden kann. Wir erhielten Zugang zu einem D-Wave 2X Adiabatic Quantum Annealer und werden experimentelle Resultate präsentieren, die die Optimierunszeit des Quantencomputers mit der Optimierungszeit eines klassischen Computers vergleichen.

Stichwörter: Anfrageoptimierung, Näherungsverfahren, Stochastische Algorithmen, parallele Verarbeitung, Quantencomputing

# Abstract

The goal of query optimization is to map a declarative query (describing data to generate) to a query plan (describing how to generate the data) with optimal execution cost. Query optimization is required to support declarative query interfaces. It is a core problem in the area of database systems and has received tremendous attention in the research community, starting with an initial publication in 1979. In this thesis, we revisit the query optimization problem. This visit is motivated by several developments that change the context of query optimization. That change is not reflected in prior literature.

First, advances in query execution platforms and processing techniques have changed the context of query optimization. Novel provisioning models and processing techniques such as Cloud computing, crowdsourcing, or approximate processing allow to trade between different execution cost metrics (e.g., execution time versus monetary execution fees in case of Cloud computing). This makes it necessary to compare alternative execution plans according to multiple cost metrics in query optimization. While this is a common scenario nowadays, the literature on query optimization with multiple cost metrics (a generalization of the classical problem variant with one execution cost metric) is surprisingly sparse. While prior methods take hours to optimize even moderately sized queries when considering multiple cost metrics, we propose a multitude of approaches to make query optimization in such scenarios practical. A second development that we address in this thesis is the availability of novel software and hardware platforms that can be exploited for optimization. We will show that integer programming solvers, massively parallel clusters (which nowadays are commonly used for query execution), and adiabatic quantum annealers enable us to solve query optimization problem instances that are far beyond the capabilities of prior approaches.

In summary, we propose seven novel approaches to query optimization that significantly increase the size of the problem instances that can be addressed (measured by the query size and by the number of considered execution cost metrics). Those novel approaches can be classified into three broad categories: moving query optimization before run time to relax constraints on optimization time, trading optimization time for relaxed optimality guarantees (leading to approximation schemes, incremental algorithms, and randomized algorithms for query optimization with multiple cost metrics), and reducing optimization time by leveraging novel software and hardware platforms (integer programming solvers, massively parallel clusters, and adiabatic quantum annealers). Those approaches are novel since they address novel problem variants of query optimization, introduced in this thesis, since they are novel for their respective problem variant (e.g., we propose the first randomized algorithm for query optimization with multiple cost metrics), or because they have never been used for optimization problems in the database domain (e.g., this is the first time that quantum computing is used to solve a database-specific optimization problem).

Key words: Query optimization, multi-objective query optimization, parametric query optimization, multiple query optimization, approximation algorithms, randomized algorithms, incremental algorithms, parallelization, integer programming, quantum computing

# Contents

# List of Figures

# List of Tables

# 1 Introduction

The goal of query optimization is to map a declarative query (describing data to generate) to an optimal query plan (describing how to generate the data) [116]. Choices related to operation order or operator implementations typically lead to myriads of alternative plans for a given query. The execution cost of an average plan is often by many orders of magnitude higher than the cost of the best plan [64]. Query optimization is therefore crucial to support declarative query interfaces efficiently. This is why relational database system integrate sophisticated query optimizer components that are the result of thousands of man years worth of work [82]. But the scope of query optimization extends beyond traditional database systems. Tools such as Hive [4] and Spark SQL [7] support SQL-like queries on top of the Hadoop [3] or Spark [6] framework. Services such as Google BigQuery [2] and Amazon RedShift [1] support SQL processing in the Cloud. All those and other systems benefit from advances in query optimization methods.

Query optimization is an NP-hard optimization problem and it is even NP-hard to find approximately optimal solutions [33]. This means that the time for finding an optimal solution (for all currently known algorithms) grows exponentially in the size of the input problem. At the same time, the query optimization problem is usually solved at run time which implies tight constraints on optimization time. Taken together, this means that the size of the queries that we can optimize in a principled fashion (i.e., with formal guarantees on finding optimal or near-optimal query plans) is in practice quite limited. Query optimization is a challenging problem that must however be solved in order to enable declarative query interfaces. This combination has led to a large body of research work in the database community, starting with a first publication in 1979 [116]. Query optimization is one of the core research problems in the database domain that keeps receiving significant attention in the community (as evidenced by dedicated tracks at the top database conferences[1]).

While there is a large body of work on query optimization, most of the existing work is based on similar assumptions as the first paper on cost-based query optimization in 1979 [116].

[1] http://www.vldb.org/2015/program/Table.html

The context of query optimization has however changed in the meantime and some of those assumptions must be revised. This change of context was not reflected appropriately in the query optimization literature prior to the work presented in this thesis.

The work in this thesis is primarily motivated by advances in query execution platforms and by advances in optimization software and hardware platforms. Execution and optimization platforms both define the context of query optimization. Advances in query execution platforms have the potential to motivate changes to the problem model in query optimization. Advances in optimization platforms represent opportunities that can be seized for making query optimization more efficient.

Query execution platforms have advanced in various ways over the past decades: flexible provisioning models such as Cloud computing and crowdsourcing allow to scale out automatic processing (Cloud computing) or human computation (crowdsourcing). Approximate processing techniques allow users to reduce query execution overhead when accepting lower result precision. At a high level of abstraction, many of those advances allow users to trade between different execution cost metrics. Cloud computing allows users to trade execution time for monetary execution fees (by adapting number and type of the computational resources rented from the Cloud provider) while approximate processing techniques allow users to trade between execution time and result precision or completeness (by adapting sample sizes). Multitenancy scenarios motivate providers to consider tradeoffs between the system resources allocated for the execution of one query (e.g., in terms of main memory, disc space, and the number of CPU cores) and its execution time. Energy consumption is becoming an important execution cost metric in addition to execution time.

In summary, there are nowadays many scenarios in which multiple execution cost metrics are important. This challenges the assumption that query plans are compared according to a single cost metric (usually execution time) that most query optimization algorithms are based upon. The literature on query optimization with multiple execution cost metrics, multi-objective query optimization, is surprisingly sparse. Existing optimization algorithms are either highly specific or highly inefficient (we discuss prior art in more detail in Section 1.1). In this dissertation, we propose first of all a broad range of techniques (including approximation algorithms, incremental algorithms, randomized algorithms, and pre-processing methods) to make query optimization with multiple diverse execution cost metrics practical.

The first algorithms for cost-based query optimization [116] were executed on a single computer and did not rely on any specialized software. Most current query optimizers still use a similar software and hardware platform. This means however that we miss opportunities to leverage advances in hardware and software platforms for optimization. Massively parallel clusters with shared-nothing architectures are nowadays commonly used for query execution. In this thesis, we will show that we can use them for query optimization as well. Software solvers for standard problems (e.g., mixed integer linear programming) have steadily improved their performance over the last decades. We will see that they enable us to treat problem

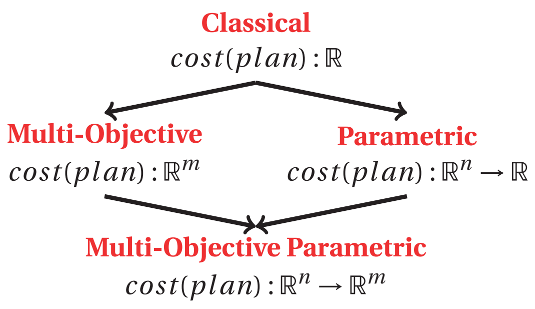

Classical
$cost(plan) : \mathbb{R}$
Multi-Objective
$cost(plan) : \mathbb{R}^m$
Parametric
$cost(plan) : \mathbb{R}^n \rightarrow \mathbb{R}$
Multi-Objective Parametric
$cost(plan) : \mathbb{R}^n \rightarrow \mathbb{R}^m$

Figure 1.1 – Query optimization variants can be classified according to how they model the cost of a single query plan (arrows represent generalizations). This thesis introduces multi-objective parametric query optimization together with a first approach, we provide the first practical algorithms for multi-objective query optimization, and we significantly extend the instance sizes that can be treated in classical and parametric query optimization.

instances in query optimization whose size exceeds the capabilities of classical algorithms by far. Finally, hardware solvers have very recently become available that exploit quantum mechanics to solve NP-hard optimization problems. We will show how to solve certain query optimization variants on such a machine. Based on a research grant giving us access to the corresponding hardware, located at NASA Research Center in California, we evaluate our approach experimentally and compare against classical computers.

The approaches that we present in this thesis are applicable to multiple query optimization variants. Some of them are classical problem variants while others are introduced in this thesis. Figure 1.1 categorizes query optimization variants based on how they model the execution cost of one single query plan. Classical query optimization considers one execution cost metric. Therefore, each query plan is associated with a scalar execution cost value (usually representing estimated execution time). Considering multiple execution cost metrics leads to multi-objective query optimization. The cost of one query plan is modeled by a cost vector where different vector components represent cost according to different metrics. Optimizing query templates (containing placeholders) instead of fully specified queries leads to parametric query optimization. In that case, the cost of a query plan is modeled by a cost function that depends on parameters with unknown values (parameters represent placeholders in the query template). The latter two query optimization variants generalize classical query optimization in different ways. Multi-objective parametric query optimization generalizes both of the latter variants by modeling the cost of a plan by a cost function mapping a multi-dimensional parameter space to a multi-dimensional cost space.

With regards to the aforementioned variants, this thesis makes the following contributions. We introduce multi-objective parametric query optimization, analyze the problem, and propose a first corresponding approach. This allows to address scenarios in query optimization that cannot be modeled using any of the other models. We are the first to make multi-objective query optimization with diverse cost metrics practical. While prior approaches take hours to optimize a single standard query (see Chapter 2), we propose a broad range of approaches that reduce optimization time to a few seconds or less. For the classical query optimization variants with one execution cost metric, we show how to exploit novel software and hardware platforms in order to treat problem instances that are far beyond the capabilities of prior approaches.

In summary, the contribution of this thesis are various approaches that very significantly extend the size of the problem instances that can be treated in query optimization. Size is not only measured by the size of the input queries but also by the number of execution cost metrics that are used to compare plans. The approaches that we propose can be categorized into three broad categories: we move query optimization before run time to relax constraints on optimization time, we reduce optimization time by relaxing optimality guarantees (leading to approximation algorithms, incremental algorithms, or randomized algorithms), or we leverage novel software and hardware platforms for optimization (massively parallel clusters, mixed integer linear programming solvers, or adiabatic quantum annealers).

The seven approaches that are presented in this thesis are derived from seven papers that have been published (with one exception) at the VLDB or at the SIGMOD conference. Some of them have additionally been invited into the "Best of VLDB" special issue of the VLDB Journal and selected for the ACM SIGMOD Research Highlight Award 2015. A list of those papers follows:

- Immanuel Trummer, Christoph Koch.
  Approximation schemes for many-objective query optimization.
  *SIGMOD 2014*.
- Immanuel Trummer, Christoph Koch.
  An incremental anytime algorithm for multi-objective query optimization.
  Talk Recording: https://www.youtube.com/watch?v=J54gVIt9UAo
  *SIGMOD 2015*.
- Immanuel Trummer, Christoph Koch.
  Multi-objective parametric query optimization.
  Talk Recording: https://www.youtube.com/watch?v=hO3IaSfFtJY
  *VLDB 2015*.
- Immanuel Trummer, Christoph Koch.
  A fast randomized algorithm for multi-objective query optimization.
  *SIGMOD 2016*.
- Immanuel Trummer, Christoph Koch.
  Parallelizing query optimization on shared-nothing architectures.
  *VLDB 2016*.
- Immanuel Trummer, Christoph Koch.
  Solving the join ordering problem via mixed integer linear programming.

http://arxiv.org/pdf/1511.02071v1.pdf, 2015.

- Immanuel Trummer, Christoph Koch.
  Multiple query optimization on the D-Wave 2X adiabatic quantum computer.
  *VLDB 2016*.

In the remainder of this introductory chapter, we will shortly discuss prior art (see Section 1.1), describe the thesis contributions in more detail (see Section 1.2), and describe the structure of this thesis (see Section 1.3).

## 1.1 State of the Art

The classical query optimization problem has been introduced in the seventies [116]. Extended variants such as parametric query optimization [78], multi-objective query optimization [107], and multiple query optimization [118] have been introduced later. In this dissertation (and in the associated paper), we also introduce the multi-objective parametric query optimization problem [139] which generalizes most of the previously proposed query optimization variants.

Query optimization algorithms can generally be classified into exhaustive algorithms, which formally guarantee to find an optimal or near-optimal solution, and heuristic or randomized algorithms. The latter type of algorithm gives up formal guarantees on generating optimal plans in order to avoid prohibitive optimization time. Modern database systems often use two different optimization algorithms in combination: an exhaustive algorithm is used by default but a randomized algorithm is used instead if queries are too large for exhaustive optimization. The Postgres database system uses for instance an exhaustive query optimization algorithm for small SQL queries and switches to a randomized algorithm for optimizing larger queries.

The general goal in query optimization is however to push back the limit on the query size starting from which randomized or heuristic algorithms have to be used. The reason is that randomized optimization can in principle lead to query plans with catastrophic execution cost. In this thesis, we present several approaches that allow to optimize significantly larger queries than prior techniques. We do so by exploiting several opportunities to speed up query optimization that have so far been overlooked. For instance, we show how many query optimization variants can be parallelized over large clusters with hundreds of nodes. Query optimization has been parallelized before but existing approaches are only able to exploit moderate degrees of parallelism in shared-memory architectures [67, 145, 68]. They do not allow to exploit massive degrees of parallelism in shared-nothing architectures. Such architectures are however often used for query execution and there is no reason not to exploit them for optimization as well.

Another opportunity that has been overlooked in query optimization is the possibility to reduce query optimization to other standard problems. We show how to reduce query optimization to mixed integer linear programming which allows to leverage extremely sophisticated solver implementations. Linear programming has already been used in the context of query optimization but only in sub-functions of traditional query optimization algorithms [59, 73].

Quantum annealing [50] is yet another opportunity to speed up optimization that has very recently become available. Quantum annealing has been recently applied to optimization problems outside of the database community [16, 109]. On the other side, there has been theoretical work examining the potential of quantum computing for certain problems in the database domain [63]. The work presented in this thesis is however unique in that it is the first time that quantum computing (implemented by the D-Wave 2X adiabatic quantum annealer) is actually applied to solve a database-specific optimization problem.

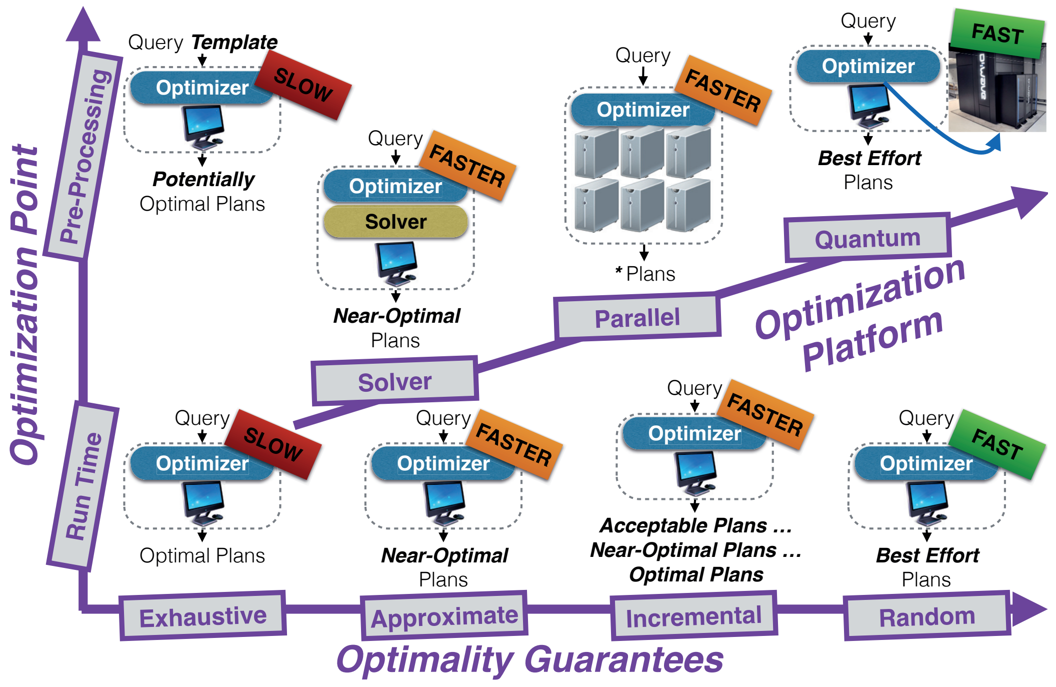
Prior to the work presented in this thesis, the literature on query optimization with multiple cost metrics has been rather sparse. Most existing approaches [13, 147] were targeted at very specific combinations of execution cost metrics and execution platforms and did not generalize to many relevant scenarios. A single algorithm would have been generic enough to deal with most relevant cost metrics [60]. This algorithm was not experimentally evaluated in the initial publication. We integrated that algorithm into the optimizer of the Postgres database system. It turns out that even the optimization of relatively simple standard queries (queries of the TPC-H benchmark) can easily take hours. While this algorithm is therefore not suitable for use in practice, we will present multiple approaches in this thesis that make generic multi-objective query optimization practical.

The coming chapters of this thesis will treat different problem variants and propose a diverse set of approaches. More detailed discussions of the state of the art concerning specific problem variants and optimization techniques can be found in the respective chapters.

## 1.2 Thesis Contributions

The main contribution of this thesis is to very significantly extend the size of problem instances that can be treated in query optimization. Problem size refers to the size of the input query but also to the number of execution cost metrics according to which alternative plans are compared. We propose various approaches that enable us to treat query optimization problem instances of a size that is far beyond the capabilities of all prior methods. Those approaches are illustrated in Figure 1.2 and can be classified into three broad categories: moving optimization before run time to relax constraints on optimization time, relaxing formal optimality guarantees to speed up optimization, and leveraging advanced software and hardware platforms to speed up optimization. We propose one or several approaches in each of those broad categories. We discuss those approaches in the following.

Query optimization usually happens at run time: a query is received by the system, the query optimizer generates an execution plan for the query which is immediately executed. Keeping optimization time low is critical in that context as it adds to the total query evaluation time. A first possibility to deal with problem instances where optimization time is high, is to move



Icons by http://www.visualpharm.com and Wallpaper FX (license: http://creativecommons.org/licenses/by/3.0/us/legalcode)

Figure 1.2 – Dissertation overview: we cope with difficult instances of the query optimization problem by either moving optimization before run time, relaxing optimality guarantees, or exploiting advanced optimization platforms. This dissertation presents one or several approaches in each of those broad categories.

optimization before run time. This does not decrease optimization time but it relaxes the constraints on optimization time: if query optimization happens before run time then higher optimization times are acceptable.

Moving query optimization before run time (see the y-axis in Figure 1.2) is possible if queries correspond to query templates that are known before run time. Query templates are not fully specified and contain placeholders that will be filled in at run time. Based on query templates, it is however possible to calculate all query plans that are potentially optimal (for at least one template instance) in a pre-processing step. Optimization at run time is thereby avoided: instead of optimization, the most suitable plan is selected out of the pre-computed plan set. Optimizing query templates instead of fully-specified queries while considering multiple execution cost metrics leads to a novel query optimization problem variant. We introduce and analyze that variant in Chapter 4 and propose a suitable optimization algorithm.

If it is not possible to move query optimization before run time then optimization time must be kept low. The x-axis in Figure 1.2 represents the possibility to reduce optimization time by relaxing optimality guarantees. In this category, we present approximation schemes (see Chapter 2) that find query plans with guaranteed near-optimal execution cost values according to multiple cost metrics. A parameter allows to trade seamlessly between optimization time and optimality guarantees. We will show, based on experiments with an extended version of the query optimizer of the Postgres database system[2], that guaranteed near-optimal plans can often be found in seconds where finding optimal plans would take hours for a given query.

In the same category, we present an incremental algorithm for query optimization with multiple execution cost metrics (see Chapter 3). This algorithm does not require to choose approximation precision beforehand. As optimization time progresses, it generates plans of increasing quality while always providing bounds on how far the current solutions are from the optima. In addition, the algorithm allows users to integrate feedback during optimization in order to guide search towards more promising parts of the search space.

Query optimization is an NP-hard optimization problem and it is NP-hard to find optimal solutions already when considering only one execution cost metric [33]. Finding optimal or guaranteed near-optimal query plans is therefore unrealistic for very large queries. Randomized algorithms become efficient by giving up any worst-case guarantees on result optimality. Prior to this thesis, randomized algorithms were available only for classical query optimization with one execution cost metric. We introduce the first randomized algorithm for multi-objective query optimization in Chapter 5. This algorithm relaxes optimality guarantees completely and treats queries of a size that is unrealistic for approximation schemes and incremental algorithms. Our randomized algorithm exploits the specific properties of the multi-objective query optimization problem and thereby outperforms classical general-purpose randomized algorithms for multi-objective optimization. While our randomized algorithm offers no worst-case guarantees on result optimality, we will see that it performs well in average.

[2]http://www.postgresql.org/

As illustrated in Figure 1.2, we assume for all previously discussed approaches that the query optimizer runs on a commodity computer without relying on specialized software. The z-axis in Figure 1.2 represents the possibility to leverage advanced optimization hardware and software platforms in order to speed up query optimization. While the approaches on the x-axis and y-axis target query optimization with multiple cost metrics, approaches on the z-axis are also applicable to classical query optimization. We discuss the approaches on the z-axis in the following.

Exploiting advanced software solvers is a first possibility to speed up query optimization. Chapter 7 describes a method that transforms instances of the classical query optimization problem into mixed integer linear (MILP) programs. This transformation replaces plan cost functions by linearized approximations. The optimal solution to the transformed problem represents therefore no optimal but a guaranteed near-optimal solution to the original problem (the cost is within a multiplicative factor that is chosen by the user or administrator). Highly sophisticated standard solvers such as CPLEX[3] can be applied to solve the resulting problems. Such solvers have steadily improved their performance over the last decades [27] (hardware independently) and we will see that they can treat significantly larger search spaces in query optimization than traditional query optimization algorithms. Furthermore, the results reported in this thesis represent only snapshots capturing the state of the art in MILP. By linking query optimization to MILP, we will automatically benefit from all future advances in this highly fruitful research domain.

Exploiting massive degrees of parallelism is another possibility to speed up query optimization. Query execution platforms are nowadays often massively parallel. If we use that parallelism for query execution, why shouldn't we use it for optimization as well? Parallel algorithms for query optimization have been proposed prior to this dissertation. Those prior algorithms are however only able to exploit very moderate degrees of parallelism. They employ a fine-grained problem decomposition method that requires parallel optimizer threads to share intermediate results. This leads to huge communication overhead when used in the shared-nothing architectures that are typical for large-scale analytics platforms. Chapter 6 describes a radically different parallelization method that decomposes the search space in the coarsest possible way. The search space is divided into a number of equal-sized partitions that corresponds to the number of optimizer threads. Those partitions can be searched independently without communication between different threads. Thereby we parallelize query optimization over large clusters with hundreds of nodes.

The proposed parallelization method is applicable to classical query optimization but also to multi-objective query optimization, parametric query optimization, and multi-objective parametric query optimization. It can be combined with several other approaches in Figure 1.2. We characterize the scenarios in which it is useful to combine several approaches in Chapter 9.

A novel hardware solver for NP-hard optimization problems has recently become available:

[3]http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

the D-Wave adiabatic quantum annealer[4]. This machine is claimed to exploit the laws of quantum mechanics to solve NP-hard optimization problems. It operates on qubits that can be in a superposition of states (1 *and* 0) that would be considered mutually exclusive according to the laws of classical physics. With a very simplifying intuition, operating on qubits allows quantum computers to explore multiple computational paths at the same time.

The D-Wave machine has been controversially discussed over the past years, focusing on the question of (i) whether or not quantum mechanics play indeed a significant role during its computation and (ii) whether its technology will lead to performance advantages over classical computers. In the meantime, strong evidence has been collected that the D-Wave machine is capable of quantum tunneling (i.e., it exploits quantum mechanics to escape from local minima during optimization) [50]. This seems to answer the first question positively, as acknowledged by MIT Professor Scott Aaronson ("this completely nails down the case for computationally-relevant collective quantum tunneling in the D-Wave machine"[5]) among others. It does however not prove performance advantages for practically relevant optimization problems. The work presented in Chapter 8 contributes to that discussion as we present experimental results evaluating the D-Wave quantum annealer on query optimization variants. Those experiments are based on a research grant giving us access to a D-Wave 2X adiabatic quantum annealer with 1097 qubits, located at NASA Ames Research Center in California.

Using the quantum annealer is challenging as it requires to translate problems into strength values of magnetic fields on and between qubits. We will see in Chapter 8 how this transformation can be accomplished for the multiple query optimization problem (a variant of classical query optimization where multiple queries are optimized according to one execution cost metric). We will also analyze the complexity of that transformation in terms of how the number of required qubits (the scarcest resource on current annealer machines) grows asymptotically in the problem dimensions. We will see that there are test cases where the quantum annealer finds near-optimal query plans by several orders of magnitude faster than a classical computer. To the best of our knowledge, this is the first time that quantum computing was used to solve a database-related optimization problem.

## 1.3 Thesis Outline

Each of the following chapters discusses one algorithm (or several similar algorithms) for a variant of the query optimization problem. Different chapters treat different problem variants. We introduce the problem model and required notations at the beginning of each chapter in order to make them self-contained. We also discuss in each chapter the prior work that relates to the specific problem variant that is treated and to the specific method that is used. More precisely, the following approaches will be discussed in the coming chapters:

[4]http://www.dwavesys.com/
[5]http://www.scottaaronson.com/blog/?p=2555

- Chapter 2 introduces approximation schemes for multi-objective query optimization that allow to gradually relax optimality guarantees to speed up optimization.
- Chapter 3 describes an incremental algorithm that divides optimization into many small incremental steps, allowing users to integrate feedback after each step. This algorithm makes query optimization an interactive process.
- Chapter 4 introduces a new query optimization problem variant: multi-objective parametric query optimization. Solving this problem allows to make optimization a preprocessing step if queries correspond to query templates that are known in advance.
- Chapter 5 describes a randomized algorithm that is tailored to the multi-objective query optimization problem. This algorithm treats significantly larger queries than the previous approaches while giving up optimality guarantees.
- Chapter 6 describes a decomposition approach that allows to parallelize classical query optimization, multi-objective query optimization, parametric query optimization, and multi-objective parametric query optimization over large clusters with hundreds of nodes. Such clusters are nowadays common for large-scale data analysis and if we use them for data processing, why shouldn't we use them for optimization?
- Chapter 7 shows how query optimization problem instances can be transformed into mixed integer linear programs. This allows to apply mature integer programming solvers to the problem which can treat significantly larger search spaces than traditional query optimization algorithms.
- Chapter 8 describes how to solve the multiple query optimization problem on a quantum computer and presents corresponding experimental results. We had access to a D-Wave 2X adiabatic quantum annealer with over 1000 qubits, located at NASA Ames Research Center in California. This is the first database-specific optimization problem that was solved using quantum computing.

In Chapter 9, we provide guidelines on how to select the right combination of the proposed optimization methods for a given scenario. We also discuss directions for future work.

# 2 Approximation

In this chapter, we will see that finding guaranteed optimal plans is hard if multiple cost metrics are considered. We will explore the potential of approximation schemes that gradually relax optimality guarantees in order to speed up optimization. We will find that guaranteed near-optimal query plans can often be found within seconds where finding guaranteed optimal query plans takes hours.

## 2.1 Introduction

Minimizing execution time is the only objective in classical query optimization [116]. Nowadays, there are however many scenarios in which additional objectives are of interest that should be considered during query optimization. This leads to the problem of multi-objective query optimization (MOQO) in which the goal is to find a query plan that realizes the best compromise between conflicting objectives. Consider the following example scenarios.

**Example 1.** *A Cloud provider lets users submit queries on data that resides in the Cloud. Queries are processed in the Cloud and users are billed according to the accumulated processing time over all nodes that participated in processing a certain query. The processing time of aggregation queries can be reduced by using sampling but this has a negative impact on result quality. From the perspective of the users, this leads to the three conflicting objectives of minimizing execution time, minimizing monetary costs, and minimizing the loss in result quality. Users specify preferences in their profiles by setting weights on different objectives, representing relative importance, and by optionally specifying constraints (e.g., an upper bound on execution time). Upon reception of a query, the Cloud provider needs to find a query plan that meets all constraints while minimizing the weighted sum over different cost metrics.*

**Example 2.** *A powerful server processes queries of multiple users concurrently. Minimizing the amount of system resources (such as buffer space, hard disk space, I/O bandwidth, and number of cores) that are dedicated for processing one specific query and minimizing that query's execution time are conflicting objectives (each specific system resource would correspond to an objective on its own). Upon reception of a query, the system must find a query plan that* *represents the best compromise between all conflicting objectives, considering weights and bounds defined by an administrator.*

The main contribution in this chapter are several MOQO algorithms that are generic enough to be applicable in a variety of scenarios (including the two scenarios outlined above) and are much more efficient than prior approaches while they formally guarantee the generation of near-optimal query plans.

### 2.1.1 State of the Art

The goal of MOQO, according to our problem model, is to find query plans that minimize a weighted sum over different cost metrics while respecting all cost bounds. This means that multiple cost metrics are finally combined into a single metric (the weighted sum); it is still not possible to reduce MOQO to single-objective query optimization and use classic optimization algorithms such as the one by Selinger [116]. Ganguly et al. have thoroughly justified why this is not possible [60]; we quickly outline the reasons in the following. Algorithms that prune plans based on a single cost metric must rely on the single-objective principle of optimality: replacing subplans (e.g., plans generating join operands) within a query plan by subplans that are better according to that cost metric cannot worsen the entire query plan according to that metric. This principle breaks when the cost metric of interest is a weighted sum over multiple metrics that are calculated according to diverse cost formulas.

**Example 3.** *Assume that each query plan is associated with a two-dimensional cost vector of the form $(t,e)$ where $t$ represents execution time in seconds and $e$ represents energy consumption in Joule. Assume one wants to minimize the weighted sum over time and energy with weight 1 for time and weight 2 for energy, i.e. the sum $t+2e$. Let $p$ be a plan that executes two subplans $p_1$ with cost vector $(7,1)$ and $p_2$ with cost vector $(6,2)$ in parallel. The cost vector of $p$ is $(7,3)$ since its execution time is the maximum over the execution times of its subplans ($7=\max(7,6)$) while its energy consumption is the sum of the energy consumptions of its subplans ($3=1+2$). Replacing $p_1$ within $p$ by another plan $p_1'$ with cost vector $(1,3)$ changes the cost vector of $p$ from $(7,3)$ to $(6,5)$. This means that the weighted cost of $p$ becomes worse (it increases from 13 to 16) even if the weighted cost of $p_1'$ (7) is better than the one of $p_1$ (9).*

The example shows that the single-objective principle of optimality can break when optimizing a weighted sum of multiple cost metrics. Based on that insight, Ganguly et al. proposed a MOQO algorithm that uses a multi-objective version of the principle of optimality [60]. This algorithm guarantees to generate optimal query plans; it is however too computationally expensive for practical use as we will show in our experiments. The algorithm by Ganguly et al. is the only MOQO algorithm that we are aware of which is generic enough to handle all objectives that were mentioned in the example scenarios before. Most existing MOQO algorithms are specific to certain combinations of objectives where the single-objective principle of optimality holds [12, 147, 81].

### 2.1.2 Contributions and Outline

We summarize our contributions before we provide details:

- Our primary contribution are **two approximation schemes** for MOQO that scale to many objectives. They formally guarantee to return near-optimal query plans while speeding up optimization by several orders of magnitude in comparison with exact algorithms.
- We **formally analyze cost formulas** of many relevant objectives in query optimization and derive several common properties. We exploit these properties to design efficient approximation schemes and believe that our observations can serve as starting point for the design of future MOQO algorithms.
- We integrated the exact MOQO algorithm by Ganguly et al. [60] and our own MOQO approximation algorithms into the Postgres optimizer and **experimentally compare** their performance on TPC-H queries.

Our approximation schemes formally guarantee the generation of query plans whose cost is within a multiplicative factor $\alpha$ of the optimum in each objective. Parameter $\alpha$ can be tuned seamlessly to trade near-optimality guarantees for lower computational optimization cost. The near-optimality guarantees distinguish our approximation schemes from pure *heuristics*, since heuristics can produce arbitrarily poor plans in the worst case. We show in our experimental evaluation that our approximation schemes reduce query optimization time from hours to seconds, comparing with an existing exact MOQO algorithm proposed by Ganguly et al. that is referred to as EXA in the following.

We discuss related work in Section 2.2 and introduce the formal model in Section 2.3. Our experimental evaluation is based on an extended version of Postgres that we describe in Section 2.4. Note that our algorithms for MOQO are not specific to Postgres and can be used within any database system. We present the first experimental evaluation of the formerly proposed EXA in Section 2.5. Our experiments relate the poor scalability of EXA to the high number of Pareto plans (i.e., plans representing an optimal tradeoff between different cost objectives) that it needs to generate. The representative-tradeoffs algorithm (RTA), that we present in Section 2.6, generates only one representative for multiple Pareto plans with similar cost tradeoffs and is therefore much more efficient than EXA. We show that most common objectives in MOQO allow to construct near-optimal plans for joining a set of tables out of near-optimal plans for joining subsets. Due to that property, RTA formally guarantees to generate near-optimal query plans if user preferences are expressed by associating objectives with weights (representing relative importance). If users can specify cost bounds in addition to weights (representing for instance a monetary budget or a deadline), RTA cannot guarantee the generation of near-optimal plans anymore and needs to be extended. We present the iterative-refinement algorithm (IRA) in Section 2.7. IRA uses RTA to generate a representative plan set in every iteration. The approximation precision is refined from one iteration to the next such that the representative plan set resembles more and more the Pareto plan set. IRA stops once it can guarantee that the generated plan set contains a near-optimal plan. A carefully selected precision refinement policy guarantees that the amount of redundant work (by repeatedly generating the same plans in different iterations) is negligible. We analyze the complexity of all presented algorithms and experimentally compare our two approximation schemes (RTA and IRA) against EXA in Section 2.8.

## 2.2 Related Work

Algorithms for **Single-Objective Query Optimization (SOQO)** are not applicable to MOQO or cannot offer any guarantees on result quality. Selinger et al. [116] presented one of the first exact algorithms for SOQO which is based on dynamic programming. **Multi-Objective Query Optimization** is the focus of this chapter. The algorithm by Ganguly et al. [60] is a generalization of the SOQO algorithm by Selinger et al. This algorithm is able to generate optimal query plans considering a multitude of objectives with diverse cost formulas. We describe it in more detail later, as we use it as baseline for our experiments.

Algorithms for MOQO have not been experimentally evaluated for more than three objectives. They are usually tailored to very specific combinations of objectives. Neither the proposed algorithms nor the underlying algorithmic ideas can be used for many-objective QO with diverse cost formulas. Allowing only additive cost formulas (and user preference functions) [147, 81] excludes for instance run time as objective in parallel execution scenarios (where time is calculated as maximum over parallel branches). The approach by Aggarwal et al. [12] is specific to the two objectives run time and confidence. Multiple objectives are only considered by selecting an optimal set of table samples prior to join ordering which does not generalize to different objectives. Optimizing different objectives separately misses optimal tradeoffs between conflicting objectives [11]. Separating join ordering and multi-objective optimization (e.g., by generating a time-optimal join tree first, and mapping join operators to sites considering multiple objectives later [61, 107]) assumes that the same join tree is optimal for all objectives. This is only valid in special cases. Papadimitriou and Yannakakis [107] present multi-objective approximation algorithms for mapping operators to sites. Their algorithms do not optimize join order and the underlying approach does not generalize to more than one bounded objective. Algorithms for multi-objective optimization of data processing workflows [124, 125, 88] are not directly applicable to MOQO. Furthermore, the proposed approaches can be classified into heuristics that do not offer near-optimality guarantees [125, 88], and exact algorithms that do not scale [124].

**Parametric Query Optimization (PQO)** assumes that cost formulas depend on parameters with uncertain values. The goal is for instance to find robust plans [18, 17] or plans that optimize expected cost [40]. PQO and MOQO share certain problem properties while subtle differences prevent us from applying PQO algorithms to MOQO problems in general. Several approaches to PQO split for instance the PQO problem into several SOQO problems [59, 72, 28] by fixing parameter values. This is not possible for MOQO since cost values, unlike parameter values, are only known once a query plan is complete and cannot be fixed in advance. Other PQO algorithms [72] directly work with cost functions instead of scalar values during bottom-up plan construction. This assumes that all parameter values can be selected out of a connected interval which is typically not the case for cost objectives such as time or disc footprint. Our work connects to **Iterative Query Optimization** since we propose iterative algorithms. Kossmann and Stocker [91] propose several iterative algorithms that break the optimization of a large table set into multiple optimization runs for smaller table sets, thereby increasing efficiency. Their algorithm is only applicable to SOQO and does not offer formal guarantees on result quality. Work on **Skyline Queries** [90] and **Optimization Queries** [65] focuses on *query processing* while we focus on *query optimization*. Our work is situated in the broader area of **Approximation Algorithms**. We use generic techniques such as *coarsening* that have been applied to other optimization problems [55, 97]; the corresponding algorithms are however not applicable to query optimization and the specific coarsening methods differ.

## 2.3 Formal Model

We represent **queries** as set of tables $Q$ that need to be joined. This model abstracts away details such as join predicates (that are however considered in the implementations of the presented algorithms). **Query plans** are characterized by the join order and the applied join and scan operators, chosen out of a set $\mathbb{J}$ of available operators. The two plans generating the inputs for the final join in a query plan $p$ are the **sub-plans** of $p$. The set $\mathbb{O}$ contains all **cost objectives** (e.g., $\mathbb{O}$ = {buffer space, execution time}); we assume that a cost model is available for every objective that allows to estimate the cost of a plan. The function $\mathbf{c}(p)$ denotes the multi-dimensional cost of a plan $p$ (bold font distinguishes vectors from scalar values). Cost values are real-valued and non-negative. Let $o \in \mathbb{O}$ an objective, then $\mathbf{c}^o$ denotes the cost for $o$ within vector $\mathbf{c}$. Let $\mathbf{W}$ a vector of non-negative weights, then the function $C_{\mathbf{W}}(\mathbf{c}) = \sum_{o\in\mathbb{O}} \mathbf{c}^o\mathbf{W}^o$ denotes the **weighted cost** of $\mathbf{c}$. Let $\mathbf{B}$ a vector of non-negative bounds (setting $\mathbf{B}^o = \infty$ means no bounds), then cost vector $\mathbf{c}$ **exceeds** the bounds if there is at least one objective $o$ with $\mathbf{c}^o > \mathbf{B}^o$. Vector **c respects** the bounds otherwise. The following two variants of the MOQO problem differ by the expressiveness of the user preference model.

**Definition 1.** ***Weighted MOQO Problem.*** *A weighted MOQO problem instance is defined by a tuple $I = \langle Q, \mathbf{W}\rangle$ where $Q$ is a query and $\mathbf{W}$ a weight vector. A solution is a query plan for $Q$. An optimal plan minimizes the weighted cost $C_{\mathbf{W}}$ over all plans for $Q$.*

**Definition 2.** ***Bounded-Weighted MOQO Problem.*** *A bounded-weighted MOQO problem instance is defined by a tuple $I = \langle Q, \mathbf{W}, \mathbf{B}\rangle$ and extends the weighted MOQO problem by a bounds vector $\mathbf{B}$. Let $P$ the set of plans for $Q$ and $P_{\mathbf{B}} \subseteq P$ the set of plans that respect $\mathbf{B}$. If $P_{\mathbf{B}}$ is non-empty, an optimal plan minimizes $C_{\mathbf{W}}$ among the plans in $P_{\mathbf{B}}$. If $P_{\mathbf{B}}$ is empty, an optimal plan minimizes $C_{\mathbf{W}}$ among the plans in $P$.*

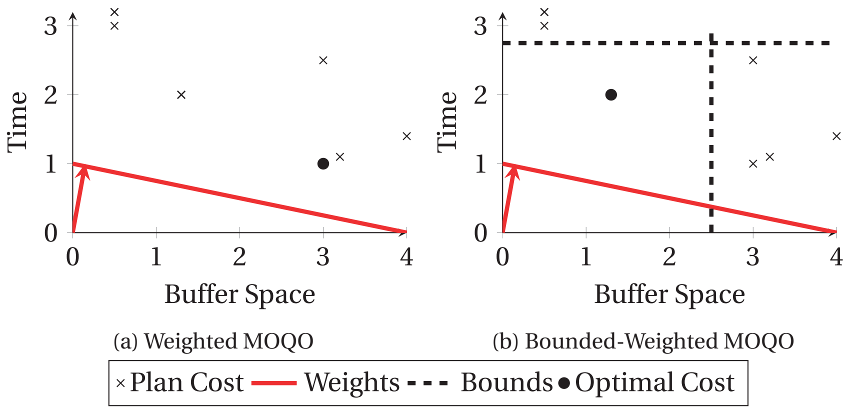


Figure 2.1 – The two MOQO problem variants

Figure 2.1a illustrates weighted MOQO. It shows cost vectors of possible query plans (considering time and buffer space as objectives) and the user-specified weights (as vector from the origin). The line orthogonal to the weight vector represents cost vectors of equal weighted cost. The optimal plan is found by shifting this line to the top until it touches the first plan cost vector. Figure 2.1b illustrates bounded-weighted MOQO. Additional cost bounds are specified and a different plan is optimal since the formerly optimal plan exceeds the bounds. We will use the set of cost vectors depicted in Figure 2.1 as **running example** throughout the chapter. The relative cost function $\rho$ measures the cost of a plan relative to an optimal plan.

**Definition 3.** ***Relative Cost.*** *The relative cost function $\rho_I$ of a weighted MOQO instance $I = \langle Q, \mathbf{W} \rangle$ judges a query plan $p$ by comparing its weighted cost to the one of an optimal plan $p^*$: $\rho_I(p) = C_{\mathbf{W}}(\mathbf{c}(p))/C_{\mathbf{W}}(\mathbf{c}(p^*))$. The relative cost function of a bounded-weighted MOQO instance $I = \langle Q, \mathbf{W}, \mathbf{B} \rangle$ is defined in the same way if no plan exists that respects $\mathbf{B}$. Otherwise, set $\rho_I(p) = \infty$ for any plan $p$ that does not respect $\mathbf{B}$ and $\rho_I(p) = C_{\mathbf{W}}(\mathbf{c}(p))/C_{\mathbf{W}}(\mathbf{c}(p^*))$ if $p$ respects $\mathbf{B}$.*

Let $\alpha \geq 1$, then an $\alpha$-**approximate solution** to a weighted MOQO or bounded-weighted MOQO instance $I$ is a plan $p$ whose relative cost is bounded by $\alpha$: $\rho_I(p) \leq \alpha$. The following classification of MOQO algorithms is based on the formal near-optimality guarantees that they offer.

**Definition 4.** ***MOQO Approximation Scheme.*** *An approximation scheme for MOQO is tuned via a user-specified precision parameter $\alpha_U$ and guarantees to generate an $\alpha_U$-approximate solution for any MOQO problem instance.*

**Definition 5.** ***Exact MOQO Algorithm.*** *An exact algorithm for MOQO guarantees to generate a 1-approximate (hence optimal) solution for any MOQO problem instance.*

The following definitions express relationships between cost vectors. A vector $\mathbf{c}_1$ **dominates** vector $\mathbf{c}_2$, denoted by $\mathbf{c}_1 \preceq \mathbf{c}_2$, if $\mathbf{c}_1$ has lower or equivalent cost than $\mathbf{c}_2$ in every objective. Vector

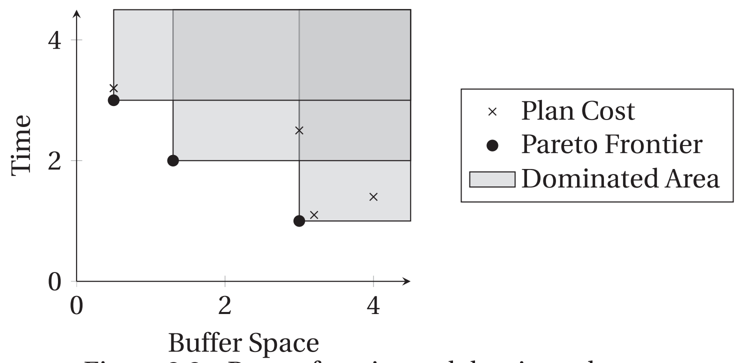
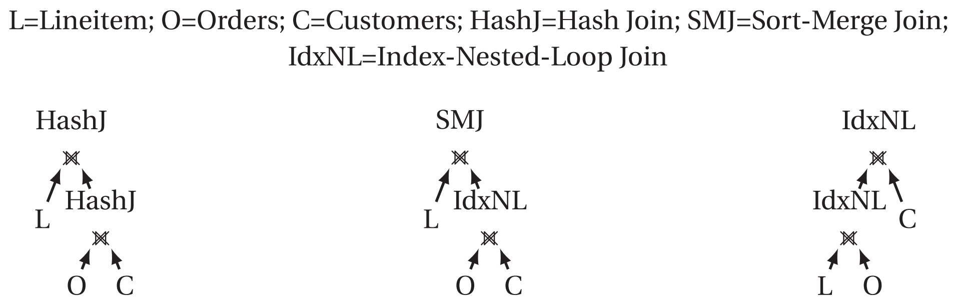


Figure 2.2 – Pareto frontier and dominated area

$\mathbf{c}_1$ **strictly dominates** $\mathbf{c}_2$, denoted by $\mathbf{c}_1 \prec \mathbf{c}_2$, if $\mathbf{c}_1 \preceq \mathbf{c}_2$ and the vectors are not equivalent ($\mathbf{c}_1 \not\equiv \mathbf{c}_2$). Vector $\mathbf{c}_1$ **approximately dominates** $\mathbf{c}_2$ with precision $\alpha$, denoted by $\mathbf{c}_1 \preceq_\alpha \mathbf{c}_2$, if the cost of $\mathbf{c}_1$ is higher at most by factor $\alpha$ in every objective, i.e. $\forall o : \mathbf{c}_1^o \leq \mathbf{c}_2^o \cdot \alpha$. A plan $p$ and its cost vector are **Pareto-optimal** for query $Q$ (short: **Pareto plan** and **Pareto vector**) if no alternative plan for $Q$ strictly dominates $p$. A **Pareto set** for $Q$ contains at least one cost-equivalent plan for each Pareto plan. The **Pareto frontier** is the set of all Pareto vectors. Figure 2.2 shows the Pareto frontier of the running example and the area that each Pareto vector dominates. An $\alpha$**-approximate Pareto set** for $Q$ contains for every Pareto plan $p^*$ a plan $p$ such that $\mathbf{c}(p) \preceq_\alpha \mathbf{c}(p^*)$. An $\alpha$**-approximate Pareto frontier** contains the cost vectors of all plans in an $\alpha$-approximate Pareto set. During complexity analysis, $j = |\mathbb{J}|$ denotes the number of operators, $l = |\mathbb{O}|$ the number of objectives, $n = |Q|$ the number of tables to join, and $m$ the maximal cardinality over all base tables in the database. Users formulate queries and have direct influence on table cardinalities. Therefore, $n$ and $m$ (and also $j$) are treated as variables during asymptotic analysis. Introducing new objectives (that cannot be derived from existing ones) requires changes to the code base and detailed experimental analysis to provide realistic cost formulas. This is typically not done by users, therefore $l$ is treated as a constant (the number of objectives is often treated as a constant when analyzing multi-objective approximation schemes [107, 55]).

## 2.4 Prototypical Implementation

We extended the Postgres system (version 9.2.4) to obtain an experimental platform for comparing MOQO algorithms. We extended the cost model, the query optimizer, and the user interface. The extended cost model supports nine objectives. The cost formulas used in the cost model are taken from prior work and are not part of our contribution. Evaluating their accuracy is beyond the scope of this chapter. We quickly describe the nine implemented cost objectives. **Total execution time** (i.e., time until all result tuples have been produced) and **startup time** (i.e., time until first result tuple is produced) are estimated according to the cost formulas already included in Postgres. Minimizing **IO load**, **CPU load**, **number of used cores**, **hard disc footprint**, and **buffer footprint** is important since it allows to increase the number of concurrent users. The five aforementioned objectives often conflict with run time since us-


L=Lineitem; O=Orders; C=Customers; HashJ=Hash Join; SMJ=Sort-Merge Join;
IdxNL=Index-Nested-Loop Join
HashJ
L
HashJ
O
C
SMJ
L
IdxNL
O
C
IdxNL
IdxNL
C
L
O

(a) Time-Optimal Plan for Bounded Tuple Loss (= 0)

(b) Additional Weight on Buffer Space Leads to Plan Without Hash Joins

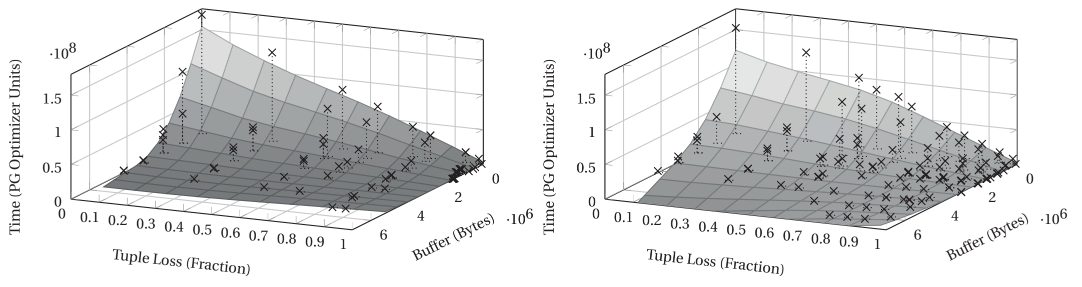
(c) Additional Bound on Startup Time Requires Using Nested-Loop Joins

Figure 2.3 – Evolution of optimal plan for TPC-H Query 3 when changing user preferences

ing more system resources can often speed up query processing. **Energy consumption** is not always correlated with time [147, 57]. Dedicating more cores to a query plan can for instance decrease execution time by parallelization while it introduces coordination overhead that results in higher total energy consumption. Energy consumption is calculated according to the cost formulas by Flach [57]. Sampling allows to trade result completeness for efficiency [66]. The **tuple loss** ratio is the expected fraction of lost result tuples due to sampling and serves as ninth objective. Joining two operands with tuple loss $a, b \in [0,1]$, the tuple loss of the result is estimated by the formula $1-(1-a)(1-b)$.

We extended the plan space of the Postgres optimizer by introducing new operators and parameterizing existing ones (we did not implement those operators in the execution engine). The extended plan space includes a parameterized sampling operator that scans between 1% and 5% of a base table. Join and sort operators are parameterized by the degree of parallelism (DOP). The DOP represents the number of cores that process the corresponding operation (up to 4 cores can be used per operation). The Postgres optimizer uses several heuristics to restrict the search space: in particular, *i)* it considers Cartesian products only in situations in which no other join is applicable, and *ii)* it optimizes different subqueries of the same query separately. We left both heuristics in place since removing them might have significant impact on performance. Not using those heuristics would make it difficult to decide whether high computational costs observed during MOQO are due to the use of multiple objectives or to the removal of the heuristics.

The original Postgres optimizer is single-objective and optimizes total execution time. We implemented all three MOQO algorithms that are discussed in this chapter: EXA, RTA, and IRA. The implementation uses the original Postgres data structures and routines wherever possible. Users can switch between the optimization algorithms and can choose the approximation precision $\alpha$ for the two approximation schemes. Users can specify weights and bounds on the different objectives. The higher the weight on some objective, the higher its relative importance. Bounds allow to specify cost limits for specific objectives (e.g., time limits or energy budgets). When optimizing a query, the optimizer tries to find a plan that minimizes



(a) Coarse-Grained Approximation ($\alpha = 2$)   (b) Fine-Grained Approximation ($\alpha = 1.25$)

▨ Pareto Surface Interpolation × Single Plan Cost

Figure 2.4 – Three-dimensional Pareto frontier approximations for TPC-H Query 5

the weighted cost among all plans that respect the bounds. Figure 2.3 shows how the optimal query plan for TPC-H query 3 changes when user preferences vary. Initially, the tuple loss is upper-bounded by zero (i.e., all result tuples must be retrieved) and all weights except the one for total execution time are set to zero. So the optimizer searches for the plan with minimal execution time among all plans that do not use sampling. Figure 2.3a shows the resulting plan. Increasing the weight on buffer footprint leads to a plan that replaces the memory-intensive Hash joins by Sort-Merge and Index-Nested-Loop (IdxNL) joins (see Figure 2.3b). Setting an additional upper bound on startup time leads to a plan that only uses IdxNL joins (see Figure 2.3c).

Users cannot make optimal choices for bounds and weights if they are not aware of the possible tradeoffs between different objectives. A user might for instance want to relax the bound on one objective, knowing that this allows significant savings in another objective. All implemented MOQO algorithms produce an (approximate) Pareto frontier as byproduct of optimization. Our prototype allows to visualize two and three dimensional projections of the Pareto frontier. Figure 2.4 shows the cost vectors of the approximate Pareto frontier for TPC-H query 5 (and an interpolation of the surface defined by those vectors), considering objectives tuple loss, buffer footprint, and total execution time. Figure 2.4a shows a coarse-grained approximation of the real Pareto frontier (with $\alpha = 2$) and Figure 2.4b a more fine-grained approximation for the same query ($\alpha = 1.25$).

## 2.5 Analysis of Exact Algorithm

Ganguly et al. [60] proposed an exact algorithm (EXA) for MOQO. This algorithm is not part of our contribution but we provide a first experimental evaluation in a many-objective scenario and a formal analysis under less optimistic assumptions than in the original publication. Algorithm 1 shows the pseudo-code of EXA (compared with the original publication, the code was slightly extended to generate bushy plans in addition to left-deep plans). EXA first calculates a Pareto plan set for query $Q$ and finally selects the optimal plan out of that set (considering weights and bounds). EXA uses dynamic programming and constructs Pareto plans for a table set out of the Pareto plans of its subsets. It is a generalization of the seminal algorithm by Selinger et al. [116], generalizing the pruning metric from one to multiple cost objectives. EXA starts by calculating Pareto plans for single tables. Plans generating the same result are compared and *pruned*, meaning that dominated plans are discarded. EXA constructs Pareto plans for table sets of increasing cardinality. To generate plans for a specific table set, EXA considers *i)* all possible splits of that set into two non-empty subsets (every split corresponds to one choice of operands for the last join), *ii)* all available join operators, and *iii)* all combinations of Pareto plans for generating the two inputs to the last join.

### 2.5.1 Experimental Analysis

We implemented EXA within the system described in Section 2.4. The implementation allows to specify timeouts (the corresponding code is not shown in Algorithm 1). If the optimization time exceeds two hours, the modified EXA finishes quickly by only generating one plan for all table sets that have not been treated so far. We experimentally evaluated EXA using the TPC-H [136] benchmark. We generated several test cases for each TPC-H query by randomly selecting subsets of objectives with a fixed cardinality out of the total set of nine objectives. All experiments were executed on a server equipped with two six core Intel Xeon processors with 2 GhZ and 128 GB of DDR3 RAM running Linux 2.6 (64 bit version). We ran five optimizer threads in parallel.

The goal of the evaluation was to answer three questions: *i)* Is the performance of EXA good enough for use in practice? *ii)* If not, how can the performance be improved? *iii)* What assumptions are realistic for the formal complexity analysis of MOQO algorithms? Figure 2.5 shows experimental results for the three metrics optimization time, allocated memory during optimization, and number of Pareto plans for the last table set that was treated completely (before a timeout occurred or before the optimization was completed). Every marker represents the arithmetic average value over 20 test cases for one specific TPC-H query and a specific number of objectives. The TPC-H queries are ordered according to the maximal number of tables that appears in any of their from-clauses. This number correlates (with several caveats[1]) with the search space size. Gray markers indicate that some test cases incurred a timeout. If a timeout occurred, then the reported values are lower bounds on the values of a completed computation.

Optimizing for one objective never requires more than 100 milliseconds per query and never consumes more than 1.7 MB of main memory. For multiple objectives, the computational cost of EXA becomes however quickly prohibitive with growing number of tables (referring to Question *i)*). EXA often reaches the timeout of two hours and allocates gigabytes of main memory during optimization. This happens already for queries joining only three tables; while the number of possible join orders is small in this case, the total search space size is

[1]The Postgres optimizer may for instance convert EXISTS predicates into joins which leads to many alternative plans even for queries with only one table in the from-clause.

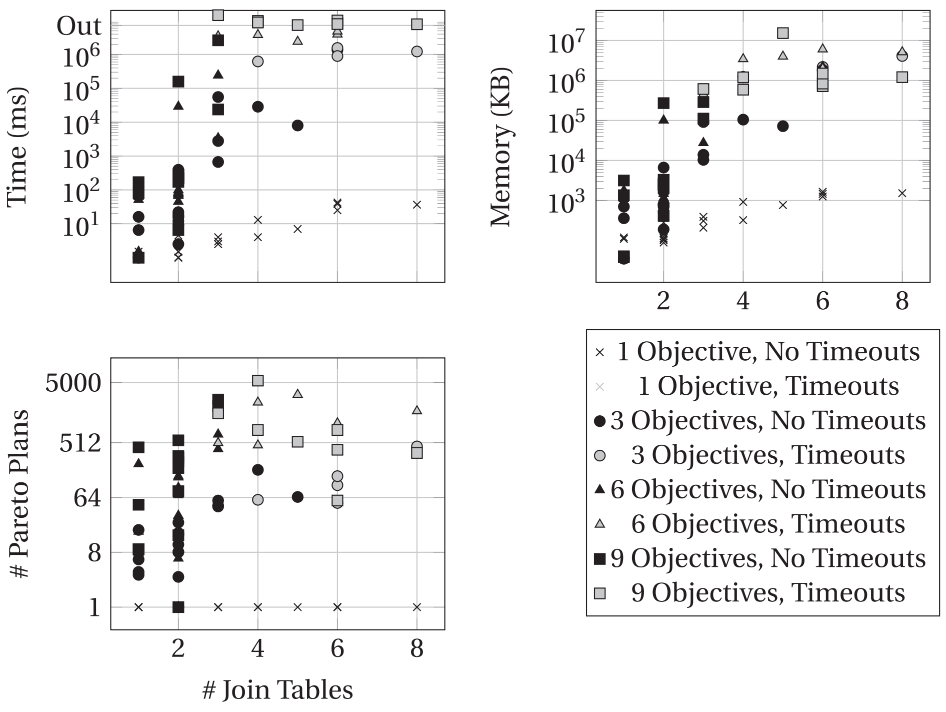


Figure 2.5 – Performance of exact algorithm on TPC-H: Prohibitive computational cost due to high number of Pareto plans (timeout at 2 hours)

already significant as over 10 different configurations are considered for the scan and for the join operator respectively (considering for instance different sample densities and different degrees of parallelism).

Figure 2.5 explains the significant difference in time and space requirements between SOQO and MOQO: The number of Pareto plans per table set is always one for SOQO but grows quickly in the number of tables (and objectives) for MOQO. The space consumption of EXA directly relates to the number of Pareto plans. The run time relates to the total number of considered plans which is much higher than the number of Pareto plans but directly correlated with it[2]. Discarding Pareto plans seems therefore the most natural way to increase efficiency (referring to Question *ii)*).

Ganguly et al. [60] used an upper bound of $2^l$ ($l$ designates the number of objectives) on the number of Pareto plans per table set for their complexity analysis of EXA. This bound derives from the optimistic assumption that different objectives are not correlated. Figure 2.5 shows that this bound is unrealistic (8, 64, and 512 are the theoretical bounds for 3, 6, and 9 objectives). The bound is a mismatch from the *quantitative* perspective (as the bound is exceeded by orders of magnitude[3]) and from the *qualitative* perspective (as the number of Pareto plans seems to correlate with the search space size while the postulated bound only depends on the number of objectives). Therefore, this bound is not used in the following complexity analysis (referring to Question *iii)*).

### 2.5.2 Formal Complexity Analysis

All query plans can be Pareto-optimal in the worst case (when considering at least two objectives). The following analysis remains unchanged under the assumption that a constant fraction of all possible plans is Pareto-optimal. If only one join operator is available, then the number of bushy plans for joining $n$ tables is given by $(2(n-1))!/(n-1)!$ [60]. If $j$ scan and join operators are available, then the number of possible plans is given by

$$\mathcal{N}_{bushy}(j,n) = j^{2n-1}(2(n-1))!/(n-1)!.$$

**Theorem 1.** *EXA has space complexity*

$$O(\mathcal{N}_{bushy}(j,n)).$$

*Proof.* Plan sets are the variables with dominant space requirements. A scan plan is represented by an operator ID and a table ID. All other plans are represented by the operator ID of the last join and pointers to the two sub-plans generating its operands. Therefore, each

[2]All plans considered for joining a set of tables are combinations of two Pareto plans; the number of considered plans therefore grows quadratically in the number of Pareto plans.

[3]We generate up to 443 Pareto plans on average when considering three objectives and up to 3157 plans when considering six objectives.

stored plan needs only $O(1)$ space. Each stored cost vector needs $O(1)$ space as well, since $l$ is a constant (see Section 2.3).

Let $Q$ the set of tables to join. EXA stores a set of Pareto plans for each non-empty subset of $Q$. The total number of stored plans is the sum of Pareto plans over all subsets. Let $k \in \{1,\ldots,|Q|\}$ and denote by $x_k$ the total number of Pareto plans, summing over all subsets of $Q$ with cardinality $k$. Each plan is Pareto-optimal in the worst case, therefore $x_k = \binom{n}{k}\mathcal{N}_{bushy}(j,k)$. It is $x_k \leq 2x_{k+1}$ for $k > 1$. Therefore, the term $x_n = \mathcal{N}_{bushy}(j,n)$ dominates. The analysis is tight since all possible plans are stored in the worst case. □

**Theorem 2.** *EXA has time complexity*

$$O(\mathcal{N}^2_{bushy}(j,n)).$$

*Proof.* Every plan is compared with all other plans that generate the same result. So the time complexity grows quadratically in the number of Pareto plans and a similar reasoning as in the proof of Theorem 1 can be applied. □

The main advantage of the single-objective Selinger algorithm [116] over a naive plan enumeration approach is that its complexity only depends on the number of table sets but not on the number of possible query plans. The preceding analysis shows that this advantage vanishes when generalizing the Selinger algorithm to multiple cost objectives (leading to EXA). The complexity of EXA is even worse than that of an approach that successively generates all possible plans while keeping only the best plan generated so far.

## 2.6 Approximating Weighted MOQO

EXA is computationally expensive since it generates all Pareto plans for each table set. We present a more efficient algorithm: the representative-tradeoffs algorithm (RTA). The new algorithm generates an approximate Pareto plan set for each table set. The cardinality of the approximate Pareto set is much smaller than the cardinality of the Pareto set. Therefore, RTA has lower computational cost than EXA while it formally guarantees to return a near-optimal plan. RTA exploits a property of the cost objectives that we call the *principle of near-optimality*. We provide a formal definition in Section 2.6.1 and show that most relevant objectives in query optimization possess that property. We describe RTA in Section 2.6.2 and prove that it produces near-optimal plans. In Section 2.6.3, we analyze its time and space complexity. We prove that its complexity is more similar to the complexity of SOQO algorithms than to the one of EXA.

### 2.6.1 Principle of Near-Optimality

The *principle of optimality* states the following in the context of MOQO [60]: If the cost of the sub-plans within a query plan decreases, then the cost of the query plan cannot increase. A formal definition follows.

**Definition 6.** ***Principle of Optimality (POO).*** *Let $P$ a query plan with sub-plans $p_L$ and $p_R$. Derive $P^*$ from $P$ by replacing $p_L$ by $p_L^*$ and $p_R$ by $p_R^*$. Then $\mathbf{c}(p_L^*) \preceq \mathbf{c}(p_L)$ and $\mathbf{c}(p_R^*) \preceq \mathbf{c}(p_R)$ together imply $\mathbf{c}(P^*) \preceq \mathbf{c}(P)$.*

The POO holds for all common cost objectives. EXA generates optimal plans as long as the POO holds. We introduce a new property in analogy to the POO. The *principle of near-optimality* intuitively states the following: If the cost of the sub-plans within a query plan increases by a certain percentage, then the cost of the query plan cannot increase by more than that percentage.

**Definition 7.** ***Principle of Near-Optimality (PONO).*** *Let $P$ a query plan with sub-plans $p_L$ and $p_R$ and pick an arbitrary $\alpha \geq 1$. Derive $P^*$ from $P$ by replacing $p_L$ by $p_L^*$ and $p_R$ by $p_R^*$. Then $\mathbf{c}(p_L^*) \preceq_\alpha \mathbf{c}(p_L)$ and $\mathbf{c}(p_R^*) \preceq_\alpha \mathbf{c}(p_R)$ together imply $\mathbf{c}(P^*) \preceq_\alpha \mathbf{c}(P)$.*

We will see that the PONO holds for the nine objectives described in Section 2.4 as well as for other common objectives. Cost formulas in query optimization are usually recursive and calculate the (estimated) cost of a plan out of the cost of its sub-plans. Different formulas apply for different objectives and for different operators. Most cost formulas only use the functions sum, maximum, minimum, and multiplication by a constant. The formula $\max(t_L, t_R) + t_M$ estimates for instance execution time of a plan whose final operation is a Sort-Merge join whose inputs are generated in parallel; the terms $t_L$ and $t_R$ represent the time for generating and sorting the left and right input operand and $t_M$ is the time for the final merge. Let $F$ any of the three binary functions sum, maximum, and minimum. Then $F(\alpha a, \alpha b) \leq \alpha F(a, b)$ for arbitrary positive operands $a, b$ and $\alpha \geq 1$. Let $F(a)$ the function that multiplies its input by a constant. Then trivially $F(\alpha a) \leq \alpha F(a)$. Therefore, the PONO holds as long as cost formulas are combined out of the four aforementioned functions (this can be proven via structural induction). The formula for tuple loss is an exception since it multiplies two factors that depend on the tuple loss in the sub-plans: The tuple loss of a plan is estimated out of the tuple loss values $a$ and $b$ of its sub plans according to the formula $F(a, b) = 1-(1-a)(1-b)$. It is $F(\alpha a, \alpha b) = \alpha(a+b) - \alpha^2 ab$. This term is upper-bounded by $\alpha(a+b-ab) = \alpha F(a, b)$ since $0 \leq a, b \leq 1$ and $\alpha \geq 1$. Note that **failure probability** is calculated according to the same formula as tuple loss (if the probabilities that single operations fail are modeled as independent Bernoulli variables). Objectives such as **monetary cost** are calculated according to similar formulas as energy consumption.

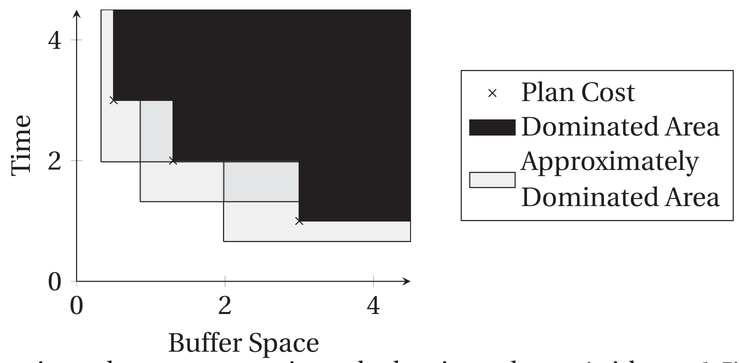


Figure 2.6 – Dominated versus approximately dominated area (with $\alpha = 1.5$) in cost space

### 2.6.2 Pseudo-Code and Near-Optimality Proof

We exploit the PONO to transform EXA into an approximation scheme for weighted MOQO. Algorithm 2 shows the parts of Algorithm 1 that need to be changed. RTA is the resulting approximation scheme. RTA takes a user-defined precision parameter $\alpha_U$ as input. It generates a plan whose weighted cost is not higher than the optimum by more than factor $\alpha_U$. We formally prove this statement later. RTA uses a different pruning function than EXA: New plans are still compared with all plans that generate the same result. But new plans are only inserted if no other plan *approximately* dominates the new one. This means that RTA tends to insert less plans than EXA. Figure 2.6 helps to illustrate this statement: EXA inserts new plans if their cost vector does not fall within the dominated area, RTA inserts new plans if their cost vector does neither fall into the dominated nor into the approximately dominated area. The following theorems exploit the PONO to show that RTA guarantees to generate near-optimal plans. They will implicitly justify the choice of the internal precision that is used during pruning.

**Theorem 3.** *RTA generates an $\alpha_i^{|Q|}$-approximate Pareto set.*

*Proof.* The proof uses induction over the number of tables $n = |Q|$. RTA examines all available access paths for single tables and generates an $\alpha_i$-approximate Pareto set. Assume RTA generates $\alpha_i^n$-approximate Pareto sets for joining $n < N$ tables (inductional assumption). Let $p^*$ an arbitrary plan for joining $n = N$ tables and $p_L^*$, $p_R^*$ the two sub-plans generating the operands for the final join in $p^*$. Due to the inductional assumption, RTA generates a plan $p_L$ producing the same result as $p_L^*$ with $\mathbf{c}(p_L) \preceq_{\alpha_i^{N-1}} \mathbf{c}(p_L^*)$, and a plan $p_R$ producing the same result as $p_R^*$ with $\mathbf{c}(p_R) \preceq_{\alpha_i^{N-1}} \mathbf{c}(p_R^*)$. The plans $p_L$ and $p_R$ can be combined into a plan $p$ that generates the same result as $p^*$ and with $\mathbf{c}(p) \preceq_{\alpha_i^{N-1}} \mathbf{c}(p^*)$, due to the PONO. RTA might discard $p$ during the final pruning step but it keeps a plan $\widetilde{p}$ with $\mathbf{c}(\widetilde{p}) \preceq_{\alpha_i} \mathbf{c}(p)$, therefore $\mathbf{c}(\widetilde{p}) \preceq_{\alpha_i^N} \mathbf{c}(p^*)$ and RTA produces an $\alpha_i^N$-approximate Pareto set. □

**Corollary 1.** *RTA is an approximation scheme for weighted MOQO.*

*Proof.* RTA generates an $\alpha_U$-approximate Pareto set according to Theorem 3 (since $\alpha_i^{|Q|} = \alpha_U$). This set contains a plan $p$ with $\mathbf{c}(p) \preceq_{\alpha_U} \mathbf{c}(p^*)$ for any optimal plan $p^*$. It is $C_{\mathbf{W}}(\mathbf{c}(p)) \leq \alpha_U \cdot C_{\mathbf{W}}(\mathbf{c}(p^*))$ for arbitrary weights $\mathbf{W}$ and $p$ is therefore an $\alpha_U$-approximate solution. □

The pruning procedure is sensitive to changes. It seems for instance tempting to reduce the number of stored plans further by discarding all plans that a newly inserted plan approximately dominates. Then the cost vectors of the stored plans can however depart more and more from the real Pareto frontier with every inserted plan. Therefore, the additional change would destroy near-optimality guarantees.

### 2.6.3 Complexity Analysis

We analyze space and time complexity. The analysis is based on the following observations.

**Observation 1.** *The cost of a plan that operates on a single table with $t$ tuples grows at most quadratically in $t$.*

**Observation 2.** *Let $F(t_L, t_R, c_L, c_L)$ the recursive formula calculating—for a specific objective and operator—the cost of a plan whose final join has inputs with cardinalities $t_L$ and $t_R$ and generation costs $c_L$ and $c_R$. Then $F$ is in*

$$O(t_L c_R + c_L + (t_L t_R)^2).$$

**Observation 3.** *There is an intrinsic constant for every objective such that the cost of all query plans for that objective is either zero or lower-bounded by that constant.*

Observations 1 and 2 trivially hold for objectives whose cost values are taken from an a-priori bounded domain such as reliability, coverage, or tuple loss (domain $[0,1]$). They clearly hold for objectives whose cost are proportional to input and output sizes[4] such as buffer or disc footprint (the maximal output cardinality of a join is $t_L t_R$ which is dominated by the term $(t_L t_R)^2$). Quicksort has quadratic worst-case complexity in the number of input tuples. It is the most expensive unary operation in our scenario, according to objectives such as time, energy, number of CPU cycles, and number of I/O operations. The (startup and total) time of a plan containing join operations can be decomposed into *i)* the time for generating the inputs to the final join, *ii)* the time for the join itself, *iii)* and the time for post-processing of the join result (e.g., sorting, hashing, materialization). The upper bound in Observation 2 contains corresponding terms, taking into account that the right (inner) operand might have to be generated several times. It does not include terms representing costs for pre-processing join inputs (e.g., hashing) as this is counted as post-processing cost of the plan generating the corresponding operand. Observation 2 can be justified similarly for objectives such as energy, number of CPU cycles, and number of I/O operations.

[4]Using size and cardinality as synonyms is a simplification since tuple (byte) size may vary. It is however realistic to assume a constant upper bound for tuple sizes (e.g., the buffer page size). Also, the analysis can be generalized.

Observation 3 clearly holds for objectives with integer cost domains such as buffer and disc footprint (bytes), CPU cycles, time (in milliseconds), and number of used cores. It also covers objectives with non-discrete value domains such as tuple loss. Tuple loss has a non-discrete value domain since—given enough tables in which we can vary the sampling rate—the tuple loss values of different plans can get arbitrarily close to each other (e.g., compare tuple loss ratio of one plan sampling 1% of every table with one that samples 2% in one table and 1% of the others, the values get closer the more tables we have). Assuming that the scan operators are parameterized by a discrete sampling rate (e.g., a percentage), there is still a gap between 0 and the minimal tuple loss ratio greater than zero. This gap does not depend on the number of tables (sampling at least one table with 99% creates a tuple loss of at least 1%). We derive a non-recursive upper bound on plan costs from our observations.

**Lemma 1.** *The cost of a plan joining $n$ tables of cardinality $m$ is bounded by $O(m^{2n})$ for every objective.*

*Proof.* Use induction over $n$. The lemma holds for $n = 1$ due to Observation 1. Assume the lemma has been proven for $n < N$ (inductional assumption). Consider a join of $N$ tables. Cost is monotone in the number of processed tuples for any objective with non-bounded domain (not for tuple loss). So every join is a Cartesian product in the worst case and that implies $(t_L t_R)^2 = m^{2N}$. The inductional assumption implies $c_L + t_L c_R \in O(m^{2N-1})$ so $(t_L t_R)^2$ remains the dominant term. □

The cost bounds allow to define an upper bound on the number of plans that RTA stores per table set.

**Lemma 2.** *RTA stores $O((n \log_{\alpha_i} m)^{l-1})$ plans per table set.*

*Proof.* Function $\delta$ maps continuous cost vectors to discrete vectors such that $\delta^o(\mathbf{c}) = \llcorner \log_{\alpha_i}(\mathbf{c}^o) \lrcorner$ for each objective $o$ and internal precision $\alpha_i$. If $\delta(\mathbf{c}_1) = \delta(\mathbf{c}_2)$ for two cost vectors $\mathbf{c}_1$ and $\mathbf{c}_2$, then $\mathbf{c}_1 \preceq_{\alpha_i} \mathbf{c}_2$ and also $\mathbf{c}_2 \preceq_{\alpha_i} \mathbf{c}_1$. This means that the cost vectors mutually approximately dominate each other. Therefore, RTA can never store two plans whose cost vectors are mapped to the same vector by $\delta$. The number of plans that have cost value zero for at least one objective is (asymptotically) dominated by the number of plans with non-zero cost values for every objective. Considering only the latter plans, their cost is lower-bounded by a constant (assume 1 without restriction of generality) and upper-bounded by a function in $O(m^{2n})$. The cardinality of the image of $\delta$ is therefore upper-bounded by $O((n \log_{\alpha_i} m)^l)$. As RTA discards strictly dominated plans, the bound tightens to $O(l(n \log_{\alpha_i} m)^{l-1})$ which equals $O((n \log_{\alpha_i} m)^{l-1})$ since $l$ is constant (see Section 2.3). □

The function $\mathcal{N}_{stored}(m, n) = (n \log_{\alpha_i} m)^{l-1}$ denotes in the following the asymptotic bound on plan set cardinalities.

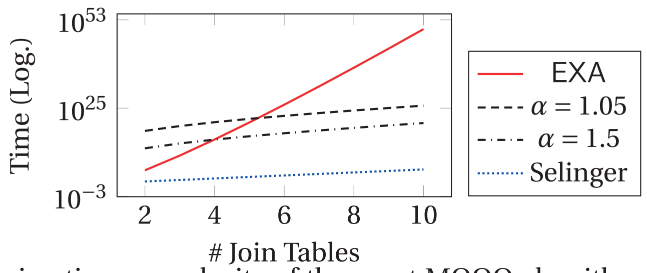


Figure 2.7 – Comparing time complexity of the exact MOQO algorithm (EXA), the MOQO approximation scheme with $\alpha = 1.05$ and $\alpha = 1.5$, and Selinger's SOQO algorithm (Setting $j = 6$, $l = 3$, and $m = 10^5$)

**Theorem 4.** *RTA has space complexity*

$$O(2^n \mathcal{N}_{stored}(m,n)).$$

*Proof.* Plan sets are the variables with dominant space consumption. Each stored plan needs only $O(1)$ space as justified in the proof of Theorem 1. Summing over all subsets of $Q$ yields the total complexity. $\square$

**Theorem 5.** *RTA has time complexity*

$$O(j3^n \mathcal{N}^3_{stored}(m,n)).$$

*Proof.* There are $O(2^k)$ possibilities of splitting a set of $k$ tables into two subsets. Every split allows to construct $O(j\mathcal{N}^2_{stored}(m,k-1))$ plans. Each newly generated plan is compared against all $O(\mathcal{N}_{stored}(m,k-1))$ plans in the set[5]. Summing time complexity over all table sets yields $\sum_{k=1..n} \binom{n}{k} 2^k j \mathcal{N}^3_{stored}(m,k) \leq j3^n \mathcal{N}^3_{stored}(m,n)$. $\square$

The time complexity is exponential in the number of tables $n$. This cannot be avoided unless $P = NP$ since finding near-optimal query plans is already NP-hard for the single-objective case [33]. The time complexity of RTA differs however only by factor $\mathcal{N}^3_{stored}(m,n)$ from the single-objective Selinger algorithm for bushy plans [140] (which has complexity $O(j3^n)$). This factor is a polynomial in number of join tables and table cardinalities. Unlike EXA, the complexity of RTA does not depend on the total number of possible plans. This lets expect significantly better scalability (see Figure 2.7 for a visual comparison).

## 2.7 Approximating Bounded MOQO

RTA finally selects an $\alpha_U$-approximate plan out of an $\alpha_U$-approximate Pareto set. This is always possible since similar cost vectors have similar weighted cost. This principle breaks when

[5]This analysis assumes that plans are compared pairwise to identify Pareto plans. Alternatively, spatial data structures [115] can be used to verify quickly if a plan's cost lie within an approximately dominated cost space area.

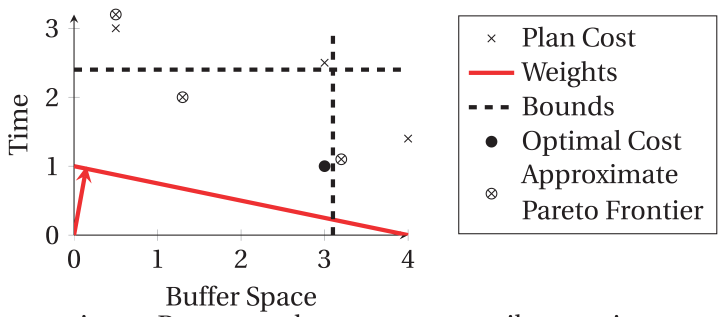


Figure 2.8 – An approximate Pareto set does not necessarily contain a near-optimal plan if bounds are considered

considering bounds in addition to weights. Even if two cost vectors are extremely similar, one of them can exceed the bounds while the other one does not. Figure 2.8 illustrates this problem. There is no $\alpha \leq \alpha_U$ except $\alpha = 1$ that guarantees a-priori that an $\alpha$-approximate Pareto set contains an $\alpha_U$-approximate plan. Choosing $\alpha = 1$ leads however to high computational cost and should be avoided (RTA corresponds to EXA if $\alpha = 1$).

Assuming that the pathological case depicted in Figure 2.8 occurs always for $\alpha > 1$ is however overly pessimistic. An $\alpha_U$-approximate Pareto set *may* very well contain an $\alpha_U$-approximate solution. We present an iterative algorithm that exploits this fact: The iterative-refinement algorithm (IRA) generates an approximate Pareto set in every iteration, starting optimistically with a coarse-grained approximation precision and refining the precision until a near-optimal plan is generated. This requires a stopping condition that detects whether an approximate Pareto set contains a near-optimal plan (without knowing the optimal plan or its cost). We present IRA and a corresponding stopping condition in Section 2.7.1. A potential drawback of an iterative approach is redundant work in different iterations. We analyze the complexity of IRA in Section 2.7.2 and show how a carefully selected precision refinement policy makes sure that the amount of redundant work is negligible. We also prove that IRA always terminates.

### 2.7.1 Pseudo-Code and Near-Optimality Proof

Algorithm 3 shows pseudo-code of IRA. IRA uses the functions FindParetoPlans and SelectBest which were already defined in Algorithm 2. IRA chooses in every iteration an approximation precision $\alpha$ and calculates an $\alpha$-approximate Pareto set. The precision gets refined from one iteration to the next. We will discuss the particular choice of precision formula in the next subsection. At the end of every iteration, IRA selects the best plan $p_{opt}$ in the current approximate Pareto set. It terminates, once that plan is guaranteed to be $\alpha_U$-optimal. The stopping condition of IRA compares $p_{opt}$ with the best plan that can be found if the bounds are slightly relaxed (i.e., multiplied by a factor). This termination condition makes sure that IRA does not terminate before it finds an $\alpha_U$-approximate plan. This implies that IRA is an approximation scheme.

**Theorem 6.** *IRA is an approximation scheme for bounded-weighted MOQO.*

*Proof.* Denote by $\mathscr{P}$ the set of plans generated in the last iteration, by $\alpha$ the precision used in the last iteration, and by $p_{opt}$ the best plan in $\mathscr{P}$. The termination condition was met in the last iteration so there is no plan $p \in \mathscr{P}$ respecting the relaxed bounds $\alpha\mathbf{B}$ with $C_{\mathbf{W}}(\mathbf{c}(p))/\alpha < C_{\mathbf{W}}(\mathbf{c}(p_{opt}))/\alpha_U$. Let $p^*$ be an optimal plan for the input query (not necessarily contained in $\mathscr{P}$). Assume first that $p^*$ respects the bounds $\mathbf{B}$. Plan set $\mathscr{P}$ contains a plan $p_R$ whose cost vector is similar to the one of $p^*$: $\mathbf{c}(p_R) \preceq_\alpha \mathbf{c}(p^*)$. The weighted cost of $p_R$ is near-optimal: $C_{\mathbf{W}}(\mathbf{c}(p_R)) \leq \alpha C_{\mathbf{W}}(\mathbf{c}(p^*))$. Plan $p_R$ can violate the bounds $\mathbf{B}$ by factor $\alpha$ but respects the relaxed bounds: $\mathbf{c}(p_R) \preceq \alpha\mathbf{B}$. Let $p$ be the best plan in $\mathscr{P}$ that respects the relaxed bounds $\alpha\mathbf{B}$, the weighted cost of $p$ is smaller or equal to the one of $p_R$. Therefore, $C_{\mathbf{W}}(\mathbf{c}(p))/\alpha$ is a lower bound on $C_{\mathbf{W}}(p^*)$. If the weighted cost of $p_{opt}$ is not higher than that by more than factor $\alpha_U$, then $p_{opt}$ is an $\alpha_U$-approximate solution. Assume now that $p^*$ does not respect the bounds $\mathbf{B}$. Then no possible plan respects the bounds and weighted cost is the only criterion. Since $\alpha \leq \alpha_U$, the set $\mathscr{P}$ must contain an $\alpha_U$-approximate solution ($p_{opt}$). □

### 2.7.2 Analysis of Refinement Policy

The formula for calculating the approximation precision $\alpha$ should satisfy several requirements. First, the formula needs to be strictly monotonically decreasing in $i$ (the number of iterations) since IRA otherwise executes unnecessary iterations that do not generate new plans. Second, it should decrease quickly enough in $i$ such that the time required by the new iteration is higher or at least comparable to the time required in all previous iterations[6]. This ensures that the amount of redundant work is small compared with the total amount of work, as IRA can generate the same plans in several iterations. Third, it should decrease as slowly as the other requirements allow; choosing a lower $\alpha$ than necessary should be avoided, since the complexity of the Pareto set approximation grows quickly in the inverse of $\alpha$. The formula $\alpha = \alpha_U^{2^{-i/(3l-3)}}$ is strictly monotonically decreasing in $i$. It also satisfies the second and third requirement as we see next. The following theorem concerns space and time complexity of the $i$-th iteration of IRA. The proof is analogous to the proofs in Section 2.6.3.

**Theorem 7.** *The $i$-th iteration of IRA has*

*space complexity* $O(2^n 2^{i/3}(n^2 \log m/\log\alpha_U)^{l-1})$,
*and time complexity* $O(j3^n 2^i(n^2 \log m/\log\alpha_U)^{3l-3})$.

Assume that the time per iteration is *proportional* to the worst-case complexity, or within a factor that does not depend on $i$ (but possibly on $n$, $m$, or $l$). Then the required time doubles from one iteration to the next, so the time of the last iteration is dominant. So the precision formula satisfies the second requirement and (approximately) the third, since decreasing iteration precision significantly slower would violate the second requirement.

**Theorem 8.** *IRA always terminates.*

[6]Memory space can be reused in the new iteration so we only consider run time in the choice of $\alpha$.

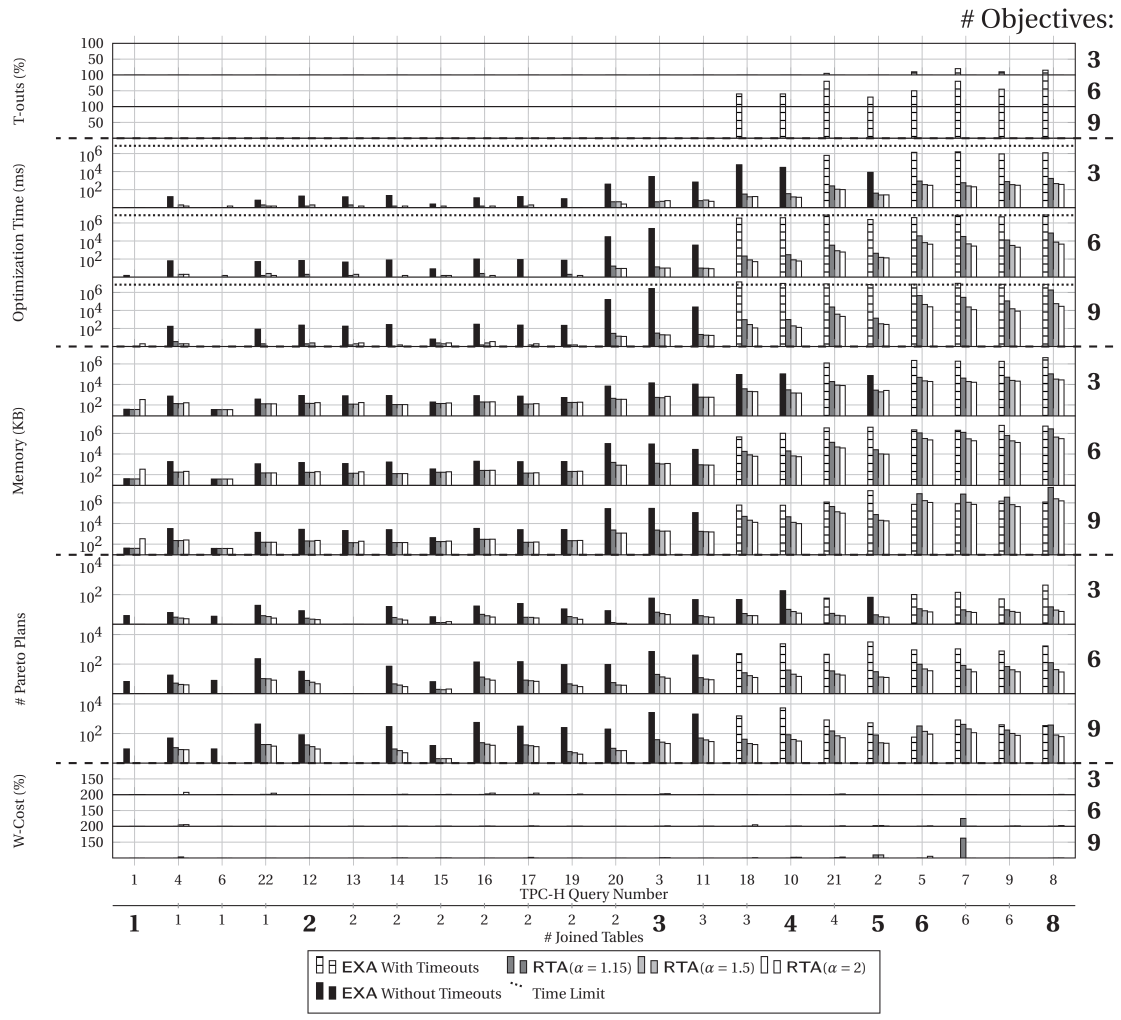


Figure 2.9 – Optimizer performance comparison for weighted MOQO using timeout of two hours

*Proof.* For a fixed bounded-weighted MOQO instance $I = \langle Q, \mathbf{W}, \mathbf{B} \rangle$ and plan space, there is only a finite number of possible query plans. Therefore, there is an $\alpha > 1$ such that no plan $p$ exists which satisfies $\mathbf{c}(p) \preceq \alpha\mathbf{B}$ but not $\mathbf{c}(p) \preceq \mathbf{B}$. The precision refinement formula is strictly monotonically decreasing in $i$ (iteration counter). So the aforementioned $\alpha$ is reached after a finite number of iterations. Then the best plan that respects the strict bounds is equivalent to the best plan that respects the relaxed bounds, so the termination condition is satisfied. $\square$

## 2.8 Experimental Evaluation

We experimentally compare the approximation schemes against EXA. The algorithms were implemented within the system described in Section 2.4. A timeout of two hours was specified, using the technique outlined in Section 2.5.1. The experiments were executed on the hardware platform described in Section 2.5.1. We generated 20 test cases for each TPC-H query and three, six, and nine objectives respectively. Every test case is characterized by a set of considered objectives (selected randomly out of the nine implemented objectives), by weights on the selected objectives (chosen randomly from $[0,1]$ with uniform distribution), and (only for bounded MOQO) by bounds on a subset of the selected objectives. Bounds for objectives with a-priori bounded value domain (e.g., tuple loss) are chosen with uniform distribution from that domain. Bounds for other objectives are chosen by multiplying the minimal possible value for a given objective and query by a factor chosen uniformly from $[1,2]$.

Figure 2.9 compares the performance of EXA and RTA with $\alpha \in \{1.15, 1.5, 2\}$. Figure 2.9 shows for each TPC-H query and each number of objectives *i)* the percentage of test cases that resulted in a timeout, and arithmetic average values for the metrics *ii)* optimization time (in milliseconds), *iii*) allocated memory during optimization (in kilobytes), *iv)* number of Pareto plans for the last table set that was treated completely (before a timeout or before optimization finished), and *v)* weighted cost of the generated plan (as percentage of the optimal value over the plans generated by all algorithms for the same test case). Queries are ordered on the x-axis according to the maximal number of tables joined in any of their from-clauses as this relates to the search space size (with the caveats mentioned in Section 2.5.1). The time limit is marked by a dotted line in the subfigure showing optimization times. The fill pattern of the bars representing results for EXA varies depending on whether EXA had at least one timeout for the corresponding query and the corresponding number of objectives (RTA did not incur any timeouts). If EXA had timeouts, then the reported values for time and memory consumption are lower bounds on the corresponding values for a completed optimizer run.

The search space size correlates with the number of tables to join, and the number of objectives influences how many plans can be pruned during optimization. Therefore, the percentage of timeouts (for EXA), the optimization time, and the memory consumption all tend to increase in the number of objectives and the number of joined tables, as long as no timeouts distort the results. EXA occasionally has timeouts already when considering only three objectives. For nine objectives, EXA is not able to solve a single test case within the limit of two hours for queries that join more than three tables. Choices related to join order, operator selection, table sample density, and parallelization create a search space of considerable size, even for only four join tables. We have seen in Section 2.5 that exact optimization takes less than 0.1 seconds despite the size of the search space, as long as only one objective is considered. Considering multiple objectives makes exact pruning however ineffective and leads to the high computational overhead of EXA. RTA is orders of magnitude faster than EXA; increasing $\alpha$ reduces optimization time and memory footprint. For nine objectives, RTA with $\alpha = 1.15$ generates for instance near-optimal plans for TPC-H query 2 within less than 1.5 seconds average time. EXA reaches the timeout of two hours for all 20 test cases.

The average quality of the plans produced by RTA is often significantly better than the worst case guarantees. Even for $\alpha = 2$, RTA generates plans with an average cost overhead of below 1% (100 times better than the theoretical bound) for 19 out of the 22 TPC-H queries. The

Postgres optimizer selects locally optimal plans for the subqueries within a query. We left this mechanism in place as justified in Section 2.4, even if it weakens the formal approximation guarantees for queries that contain subqueries (TPC-H queries 2, 4, 7, 8, 9, 11, 13, 15, 16, 17, 18, 20, 21, 22). In practice, the approximation guarantees were only violated in one case (TPC-H query 7) and only for specific choices of $\alpha$ ($\alpha = 1.15$).

Figure 2.10 shows the results for bounded MOQO. EXA is compared against IRA (instead of RTA) since only IRA guarantees to generate query plans that respect all hard bounds if such plans exist. Optimization always considers all nine objectives while the number of bounds varies between three and nine. Figure 2.10 reports the number of iterations (instead of the number of Pareto plans), the reported numbers for memory consumption refer to the memory reserved in the last iteration (memory that was allocated before can be reused). The performance of EXA is insensitive to the number of bounds. The performance of IRA varies with the number of bounds: Time and memory consumption tend to be higher when hard bounds are specified. This can be seen by comparing Figure 2.10 with Figure 2.9, as IRA behaves exactly like RTA if no bounds are specified. The reason is that IRA may have to choose a much smaller internal approximation factor than RTA, in order to verify if the best generated query plan is near-optimal among all plans respecting the bounds. The performance gap between approximate and exact MOQO is still significant: Summing over all test cases for bounded MOQO, EXA had 464 timeouts while each IRA instance had at most 4 timeouts. The total optimization time was more than 1200 hours for EXA and less than 15 hours for IRA with $\alpha = 1.15$. The number of iterations of IRA increases sometimes with the user-defined approximation factor. If hard bounds are set then the internal approximation precision required to guarantee a near-optimal plan does not necessarily correlate with the user-defined precision. However, even if the number of iterations increases, the total optimization time is not influenced significantly (except for queries with very low total optimization time where overhead by repeated query preprocessing is non-negligible). This was the goal of our precision refinement policy.

## 2.9 Conclusion

Our MOQO approximation schemes find guaranteed near-optimal plans within seconds where exhaustive optimization takes hours. We analyzed the cost formulas of typical cost metrics in MOQO and identified common properties. We believe that our findings can be exploited for design and analysis of future MOQO algorithms.

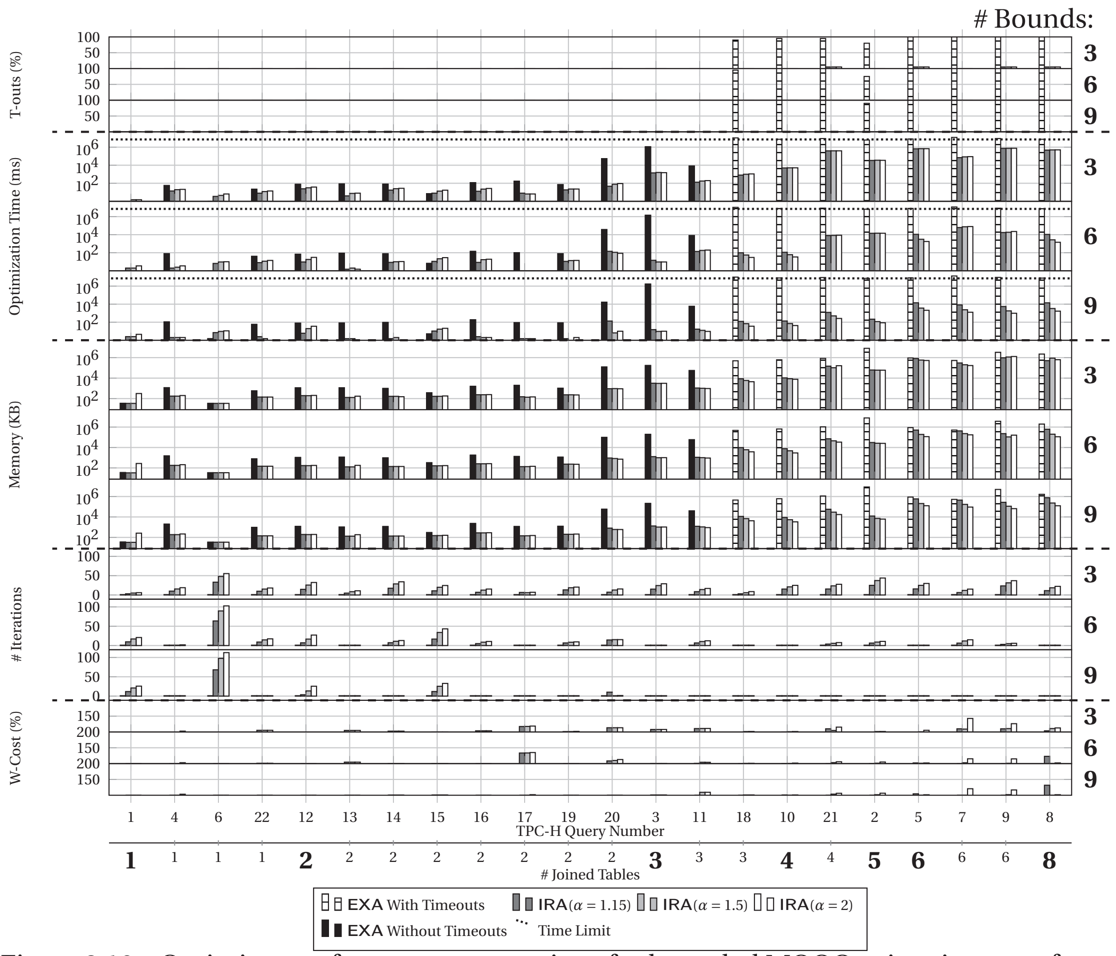


Figure 2.10 – Optimizer performance comparison for bounded MOQO using timeout of two hours

```
// Find best plan for query $Q$, weights $\mathbf{W}$, bounds $\mathbf{B}$
function ExactMOQO($Q, \mathbf{W}, \mathbf{B}$)
    // Find Pareto plan set for $Q$
    $\mathscr{P} \leftarrow$ FindParetoPlans($Q$)
    // Return best plan out of Pareto plans
    return SelectBest($\mathscr{P}, \mathbf{W}, \mathbf{B}$)
end function

// Find Pareto plan set for query $Q$
function FindParetoPlans($Q$)
    // Calculate plans for singleton sets
    for all $q \in Q$ do
        $\mathscr{P}^q \leftarrow \emptyset$
        for all $p_N$ access path for $q$ do
            Prune($\mathscr{P}^q, p_N$)
        end for
    end for
    // Consider table sets of increasing cardinality
    for all $k \in 2..|Q|$ do
        for all $q \subseteq Q : |q| = k$ do
            $\mathscr{P}^q \leftarrow \emptyset$
            // For all possible splits of set $q$
            for all $q_1, q_2 \subset q : q_1 \dot{\cup} q_2 = q$ do
                // For all sub-plans and operators
                for all $p_1 \in \mathscr{P}^{q_1}, p_2 \in \mathscr{P}^{q_2}, j \in \mathbb{J}$ do
                    // Construct new plan out of sub-plans
                    $p_N \leftarrow$ Combine($j, p_1, p_2$)
                    // Prune with new plan
                    Prune($\mathscr{P}^q, p_N$)
                end for
            end for
        end for
    end for
    return $\mathscr{P}^Q$
end function

// Prune plan set $\mathscr{P}$ with new plan $p_N$
procedure Prune($\mathscr{P}, p_N$)
    // Check whether new plan useful
    if $\neg\exists p \in \mathscr{P} : \mathbf{c}(p) \preceq \mathbf{c}(p_N)$ then
        // Delete dominated plans
        $\mathscr{P} \leftarrow \{p \in \mathscr{P} \mid \neg(\mathbf{c}(p_N) \preceq \mathbf{c}(p))\}$
        // Insert new plan
        $\mathscr{P} \leftarrow \mathscr{P} \cup \{p_N\}$
    end if
end procedure

// Select best plan in $\mathscr{P}$ for weights $\mathbf{W}$ and bounds $\mathbf{B}$
function SelectBest($\mathscr{P}, \mathbf{W}, \mathbf{B}$)
    $P_{\mathbf{B}} \leftarrow \{p \in P \mid \mathbf{c}(p) \preceq \mathbf{B}\}$
    if $P_{\mathbf{B}} \neq \emptyset$ then
        return $\arg\min[p \in P_{\mathbf{B}}] C_{\mathbf{W}}(\mathbf{c}(p))$
    else
        return $\arg\min[p \in P] C_{\mathbf{W}}(\mathbf{c}(p))$
    end if
end function
```

Algorithm 1 – Exact algorithm for MOQO

```
// Find $\alpha_U$-approximate plan for query $Q$, weights $\mathbf{W}$
function RTA($Q, \mathbf{W}, \alpha_U$)
    // Find $\alpha_U$-approximate Pareto plan set
    $\mathscr{P} \leftarrow$ FindParetoPlans($Q, \alpha_U$)
    // Return best plan in $\mathscr{P}$ for infinite bounds
    return SelectBest($\mathscr{P}, \mathbf{W}, \infty$)
end function

// Find $\alpha_U$-approximate Pareto plan set
function FINDPARETOPLANS($Q, \alpha_U$)
    // Derive internal precision from $\alpha_U$
    $\alpha_i \leftarrow \sqrt[|Q|]{\alpha_U}$
    ...
    [13] // Prune access paths for single tables
    Prune($\mathscr{P}^q, p_N, \alpha_i$)
    ...
    [25] // Prune plans for non-singleton table sets
    Prune($\mathscr{P}^q, p_N, \alpha_i$)
    ...
end function
// Prune set $\mathscr{P}$ with plan $p_N$ using precision $\alpha_i$
procedure PRUNE($\mathscr{P}, p_N, \alpha_i$)
    // Check whether new plan useful
    if $\neg\exists p \in \mathscr{P} : \mathbf{c}(p) \preceq_{\alpha_i} \mathbf{c}(p_N)$ then
    ...
    end if
end procedure
```

Algorithm 2 – The Representative-Tradeoffs Algorithm: An approximation scheme for Weighted MOQO. The code shows only the differences to Algorithm 1.

```
1:  // Find $\alpha_U$-approximate plan for query $Q$,
2:  // weights $\mathbf{W}$, bounds $\mathbf{B}$
3:  **function** IRA($Q, \mathbf{W}, \mathbf{B}, \alpha_U$)
4:      $i \leftarrow 0$ // Initialize iteration counter
5:      **repeat**
6:          $i \leftarrow i + 1$
7:          // Choose precision for this iteration
8:          $\alpha \leftarrow \alpha_U^{2^{-i/(3l-3)}}$
9:          // Find $\alpha$-approximate Pareto plan set
10:         $\mathscr{P} \leftarrow$ FindParetoPlans($Q, \alpha$)
11:         // Select best plan in $\mathscr{P}$
12:         $p_{opt} \leftarrow$SelectBest($\mathscr{P}, \mathbf{W}, \mathbf{B}$)
13:     **until** $\nexists p \in \mathscr{P} : \mathbf{c}(p) \preceq \alpha\mathbf{B} \wedge \frac{C_{\mathbf{W}}(\mathbf{c}(p))}{\alpha} < \frac{C_{\mathbf{W}}(\mathbf{c}(p_{opt}))}{\alpha_U}$
14:     **return** $p_{opt}$
15: **end function**
```

Algorithm 3 – The Iterative-Refinement Algorithm: An Approximation Scheme for Bounded-Weighted MOQO. The Code Uses Sub-Functions From Algorithm 2.

# 3 Incrementalization

The approximation schemes presented in the last chapter make multi-objective query optimization feasible for medium-sized queries. They require however users to specify their preferences before optimization starts. For users, it is often more convenient to select preferred execution cost tradeoffs in an interactive process. The algorithms presented in the last chapter are not suitable to support interactive query optimization. In this chapter, we will see an incremental anytime algorithm that divides optimization into many small incremental steps. This algorithm enables responsive user interfaces where users may integrate their preferences after each incremental step, thereby leading optimization quickly towards interesting parts of the cost space.

## 3.1 Introduction

Classical query optimization considers only one cost metric for query plans and aims at finding a plan with minimal cost [116]. 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 [107, 137, 139]. 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.

The approach presented in the last chapter 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 generally have troubles accurately expressing their preferences indirectly in a multi-objective scenario without having prior knowledge of the available tradeoffs [146]. 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.

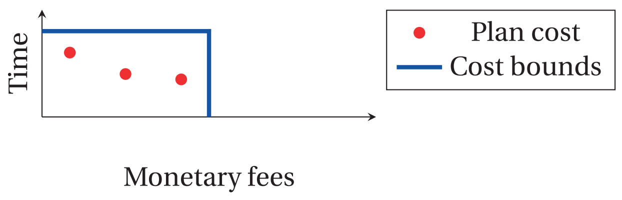
**Example 4.** *In cloud computing, there is a tradeoff between execution time and fees as buying more resources can speed up execution. Users performing SQL processing in the cloud can benefit from a visualization of available cost tradeoffs before they select a query plan for execution.*

**Example 5.** *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.*

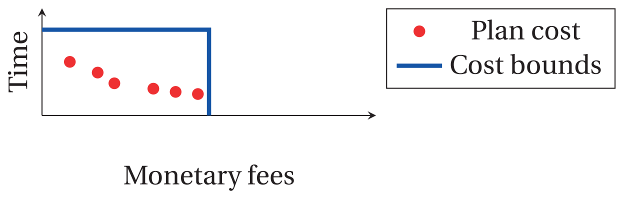
It is not necessary to make users aware of all alternative query plans for their query. It is sufficient if users have an overview of the *Pareto-Optimal* plans; a plan is Pareto-optimal if no alternative plan has better cost according to all cost metrics at the same time (this definition is slightly 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 the user query as input. The problem is that the number of Pareto plans might be extremely large already for medium-sized queries. We have seen in Chapter 2 that calculating all Pareto-optimal plans is often not realistic within a reasonable time frame. 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. 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 3.1 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.

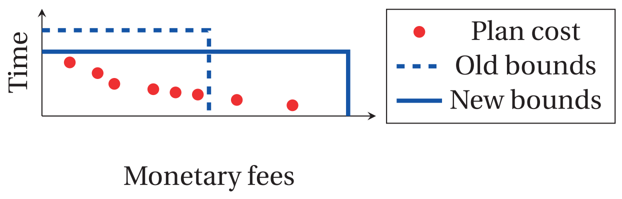
The algorithms that we have seen in Chapter 2 are however ill-suited to be used within such



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

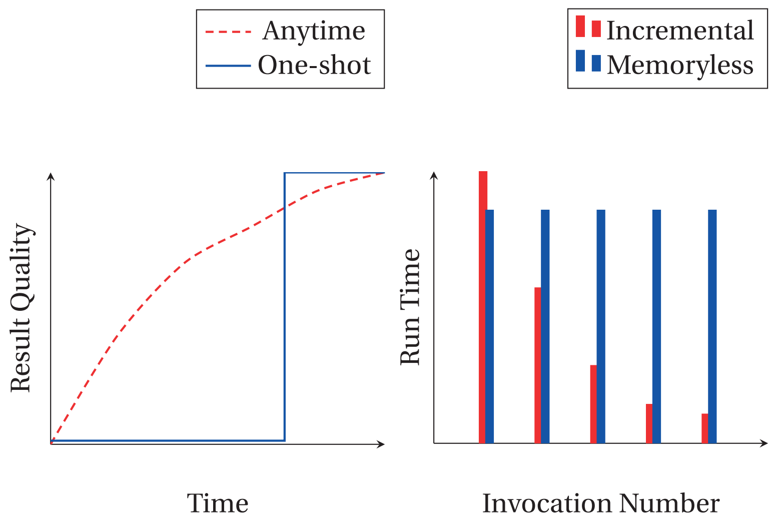


(b) The approximation is continuously refined without user interaction



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

Figure 3.1 – Example interaction between user and interactive anytime optimizer: the user selects a query plan by finally clicking on the desired cost tradeoff.



(a) Anytime versus one-shot algorithms (b) Incremental versus memoryless algorithms

Figure 3.2 – Incremental anytime algorithms

an interface for several reasons. First, they require to specify a target precision in advance and return results only once a full result 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 3.2a 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 3.2b illustrates the difference between incremental and memoryless algorithms.

The original scientific contribution of this chapter 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 chapter:

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 chapter is organized as follows. Section 3.2 discusses related work. Section 3.3 introduces the formal model that is used throughout the remainder of the chapter. Section 3.4 discusses the incremental anytime algorithm for interactive MOQO in detail. Section 3.5 proves correctness of the algorithm, analyzes its space complexity, the time complexity of a single invocation, and the amortized time of several invocations. Section 3.6 contains experimental results for TPC-H queries; the presented algorithm was implemented on top of the Postgres optimizer.

## 3.2 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 [116] 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 Chapter 2. MOQO algorithms were proposed in the context of the Mariposa system [133] 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 [107]. 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 chapter. It has been shown that even single-objective query optimization cannot be approximated in polynomial time if join order is optimized [33]; those results apply to MOQO as well since MOQO is a generalization of single-objective query optimization. Ganguly et al. [60] 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 (see Chapter 2). In Chapter 2, 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 [146]; 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 3.4. During formal analysis (see Section 3.5) and experimental evaluation (see Section 3.6), 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 [147, 12]; while such special-purpose algorithms achieve good performance, they break when taking into account additional cost metrics [137]. The algorithm proposed in this chapter 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.

Anytime algorithms are algorithms whose result quality improves gradually as computation time progresses [149]; 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* [144, 69] while we use them for *query optimization*. Chaudhuri motivated the development of anytime algorithms for classical query optimization [34]. 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 [15]). 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 [28]; 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 Chapter 4. The algorithm presented in this chapter is an approximation scheme [87]: 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 described in the previous chapter but they are neither anytime algorithms nor incremental. Our algorithm is iterative which connects it to other iterative query optimization algorithms [92, 137]. The algorithm proposed by Kossmann and Stocker [92] is however only applicable to single-objective query optimization while our iterative algorithm from the previous chapter 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.

## 3.3 Model

Our notation is similar to the one used in the previous chapter. Nevertheless, we introduce notation from scratch to make the current chapter self-contained.

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 3.4 but we outline in Section 3.4.3 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 [116]. 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 3.4 supports the same class of cost metrics as the one described in the previous chapter; 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 3.5. The focus of this dissertation 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 $P^*_\alpha$ of $P$ is an $\alpha$-*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 $\alpha$ is chosen, the lower the approximation precision and the smaller the corresponding approximate Pareto plan set can be. We derive size bounds in Section 3.5.

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 $\alpha$-*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 $\alpha$, and cost bounds $\mathbf{b}$. The result is a $\alpha$-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 the ones made in the previous chapter.

## 3.4 Description of Algorithm

We describe the Incremental Anytime MOQO Algorithm, short IAMA, for interactive MOQO in this section. The algorithms presented in the previous chapter 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 sub-function 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 [116] 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 3.5 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

[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.

resolution level or lower. We illustrate how our algorithm works by means of a highly simplified example before providing details.

**Example 6.** *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 3.4.1 describes the main control loop and Section 3.4.2 discusses the pseudo-code of the incremental optimizer. Section 3.4.3 proposes finally several extensions.

### 3.4.1 Main Control Loop

Algorithm 4 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 4 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 $\infty$, 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 3.1 from Section 3.1 for an example: the resolution increases from Figure 3.1a to Figure 3.1b). 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 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

```
// Generate Pareto plan set of increasing resolution
// for query $Q$ until user selects query plan
function IncAnyMOQO($Q$)
    // Initialize bounds and resolution
    $\mathbf{b} \leftarrow$ default bounds
    $r \leftarrow 0$
    // Fill in scan plans for single tables
    for $q \in Q, p \in$ScanPlans($q$) do
        Prune($Res^q, Cand^q, \mathbf{b}, r, p$)
    end for
    // Main control loop
    repeat
        // Generate more plans
        Optimize($Q, Res, Cand, \mathbf{b}, r$)
        // Visualize cost of known plans
        Visualize($Res^Q[\mathbf{0}..\mathbf{b}, 0..r]$)
        // Update bounds or refine resolution
        if User changed bounds then
            $\mathbf{b} \leftarrow$ user-specified bounds
            $r \leftarrow 0$
        else
            // Refine resolution until $r_M$ is reached
            $r \leftarrow \min(r_M, r+1)$
        end if
    until User selects plan $p$
    // Return result plan
    return $p$
end function
```

Algorithm 4 – Main function for interactive query optimization: processes user input, visualizes plan cost, and invokes incremental optimization procedure.

range queries was compiled by Bentley and Friedman [24]. 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 3.5.4). 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 4. 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 sub-function can alter the state of $Res$ and $Cand$. Procedure Visualize visual-

```
// Generate plans for query $Q$, insert them into result
// set $Res$ if they are relevant for current resolution $r$
// and bounds $\mathbf{b}$ or insert them into candidate set $Cand$
// if they might become relevant later.
procedure OPTIMIZE($Q, Res, Cand, \mathbf{b}, r$)
    // Check candidate plans
    for $q \subseteq Q$ do
        for $p_C \in Cand^q[\mathbf{0}..\mathbf{b}, 0..r]$ do
            $Cand^q \leftarrow Cand^q \setminus \{p_C\}$
            PRUNE($Res^q, Cand^q, \mathbf{b}, r, p_C$)
        end for
    end for
    // Generate plans using fresh candidates
    for $k \leftarrow 2$ to $|Q|$ do
        for $q \subseteq Q : |q| = k$ do
            for $q_1 \subset Q : q_1 \neq \emptyset; q_2 \leftarrow Q \setminus q_1$ do
                for $p_F \in$FRESH($Res^{q_1}, Res^{q_2}, \mathbf{b}, r$) do
                    PRUNE($Res^q, Cand^q, \mathbf{b}, r, p_F$)
                end for
            end for
        end for
    end for
end procedure
```

Algorithm 5 – Incremental optimization algorithm for multi-objective query optimization.

izes 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 $Res^Q[\mathbf{0}..\mathbf{b}, 0..r]$.

### 3.4.2 Incremental Optimizer

Algorithm 5 is the incremental optimizer procedure that is invoked in each iteration of the main loop (lines 12 to 25 in Algorithm 4). 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: the non-incremental MOQO algorithm from the last chapter 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 5 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 5 (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 [116], 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 6 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 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 $\alpha_r$ 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 $\alpha_r$ 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 3.6. 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 $\alpha_r$ at least. The higher the factor $\alpha_r$, 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 3.5. We also show in Section 3.5 that an invocation of the optimizer function with resolution $r$ yields an $\alpha_r^n$-approximate Pareto plan set, where $n = |Q|$ designates the number

```
// Insert plan $p$ for query $q$ into result set $Res$ if $p$ is
// relevant for current resolution $r$ and bounds $\mathbf{b}$.
// Insert $p$ into candidate set $Cand$ if $p$ could
// become relevant later.
procedure PRUNE($Res^q, Cand^q, \mathbf{b}, r, p$)
    // Compare $p$ with alternative plans and bounds
    if $\exists p_A \in Res^q[\mathbf{0}..\mathbf{b}, 0..r] : \mathbf{c}(p_A) \preceq \alpha_r \cdot \mathbf{c}(p)$ then
        // $p_A$ approximates $p$ for resolution $r$
        // → keep $p$ as candidate for higher resolutions
        if $r < r_M$ then
            INSERT($Cand^q[\mathbf{c}(p), r+1], p$)
        end if
    else if $\mathbf{c}(p) \not\preceq \mathbf{b}$ then
        // Cost of $p$ exceeds the bounds
        // → keep $p$ as candidate for different bounds
        INSERT($Cand^q[\mathbf{c}(p), r], p$)
    else
        // $p$ is immediately relevant
        // → add $p$ to result plan set
        INSERT($Res^q[\mathbf{c}(p), r], p$)
    end if
end procedure
// Given two sets of sub-plans $Res^{q_1}$ and $Res^{q_2}$, filter
// to relevant plans for resolution $r$ and bounds $\mathbf{b}$ and
// generate all fresh combinations of sub-plans.
function FRESH($Res^{q_1}, Res^{q_2}, \mathbf{b}, r$)
    // Filter to relevant sub-plans
    $P_1 \leftarrow Res^{q_1}[\mathbf{0}..\mathbf{b}, 0..r]$
    $P_2 \leftarrow Res^{q_2}[\mathbf{0}..\mathbf{b}, 0..r]$
    // Generate relevant sub-plan pairs
    $pairs \leftarrow \Delta P_1 \times (P_2 \setminus \Delta P_2)$
    $pairs \leftarrow pairs \cup ((P_1 \setminus \Delta P_1) \times \Delta P_2)$
    $pairs \leftarrow pairs \cup (\Delta P_1 \times \Delta P_2)$
    // Generate fresh plans
    $fresh \leftarrow \emptyset$
    for $\langle p_1, p_2 \rangle \in pairs$ :ISFRESH($p_1, p_2$) do
        $fresh \leftarrow fresh \cup \{p_1 \bowtie p_2\}$
    end for
    return $fresh$
end function
```

Algorithm 6 – Sub-functions of the optimization procedure.

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 $\alpha_r$ and the result precision allows to choose the factors $\alpha_r$ such that a desired target precision is still reached for the maximal expected number of tables; alternatively, the choice of $\alpha_r$ 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 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 the approximation schemes from the previous chapter which always keep the result plan sets as small as possible. 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 3.5.2); 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 $\Delta S$ 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 $\Delta S$ (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 $\Delta S$ and $(S \setminus \Delta S)$ 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.

### 3.4.3 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 3.6 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 6). 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* [116] 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 [116]. The seminal paper by Selinger [116] 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.

## 3.5 Formal Analysis

We analyze the algorithm presented in Section 3.4: More precisely, we analyze the optimizer sub-function that is represented in Algorithm 5. Section 3.5.1 proves formal worst-case guarantees on how closely the result plan sets, produced by the optimizer, approximate the real Pareto plan set. Section 3.5.2 analyzes space complexity and Section 3.5.3 analyzes the time complexity of a single optimizer invocation. In Section 3.5.4, we analyze the amortized time complexity of several consecutive optimizer invocations for the same query.

### 3.5.1 Result Precision

The following analysis is based on the *Principle of Near-Optimality*(PONO) for MOQO, introduced in the previous chapter, 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 8** (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.

**Theorem 9.** *After invoking* OPTIMIZE *with bounds $\mathbf{b}$ and resolution $r$ for query $Q$, $Res^q[\mathbf{0}..\mathbf{b}, 0..r]$ contains an $\alpha_r$-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 $\alpha_r$-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 $\alpha_r$-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 $\alpha_r$-approximate $\mathbf{b}$-bounded Pareto plan set. □

**Theorem 10.** *After invoking* OPTIMIZE *with bounds $\mathbf{b}$ and resolution $r$ for query $Q$, $Res^q[\mathbf{0}..\mathbf{b}, 0..r]$ contains an $\alpha_r^k$-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 9 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

[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.

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 $\alpha_r$ and the approximation quality of the result plan sets allows to choose the factor $\alpha_{r_M}$ for the maximal resolution in function of the desired target precision.

### 3.5.2 Space Consumption

The optimizer (meaning procedure OPTIMIZE, see Algorithm 5) 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 3.** *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, as shown in the previous chapter. 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 $\alpha_r$ 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 4.** *$Cand^q$ 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 sub-plan 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 11.** *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 sub-plans generating the operands for the final join. Plan cost vectors have constant space consumption since $l$ is treated as a constant (see Section 3.3). 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 [24]. 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. □

### 3.5.3 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 [24]: 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

[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.

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 5.** *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 **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 3.3). Adding the new plan takes constant time. □

**Lemma 6.** *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 12.** *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 **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 $O(3^n rpt^3(n,\mathbf{b},r))$. □

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.

### 3.5.4 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 $\Delta$ 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 7.** *Each possible plan is generated at most once.*

*Proof.* Scan plans are only generated before the main loop of Algorithm 4 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 8.** *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 9.** *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 13.** *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 5 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 3.5.2), as each plan and plan pair is generated only once (Lemmata 7 and 8), and as each plan is retrieved at most $r_M + 1$ times (Lemma 9), 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.

## 3.6 Experimental Evaluation

Section 3.6.1 describes and justifies the experimental setup and Section 3.6.2 discusses the experimental results.

### 3.6.1 Experimental Setup

We evaluate an incremental anytime algorithm for MOQO, its simplified pseudo-code was presented in Section 3.4, in comparison with two baselines: the *memoryless algorithm* is equivalent to the iterative MOQO algorithm proposed in the previous chapter except that we use a different precision refinement policy, the *one-shot algorithm* corresponds to the non-iterative MOQO algorithm presented in Chapter 2 as well. 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 $\infty$.

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 Chapter 2. 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 $\alpha_T$ and for the precision step $\alpha_S$. All experiments were executed on a MacBook air with 4 GB RAM and an Intel Core i5 processor with 1.4 GHz.

### 3.6.2 Experimental Results

Figure 3.3 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 3.5.1 and on the fact that TPC-H queries have at most eight tables. Hence the costs of the result query

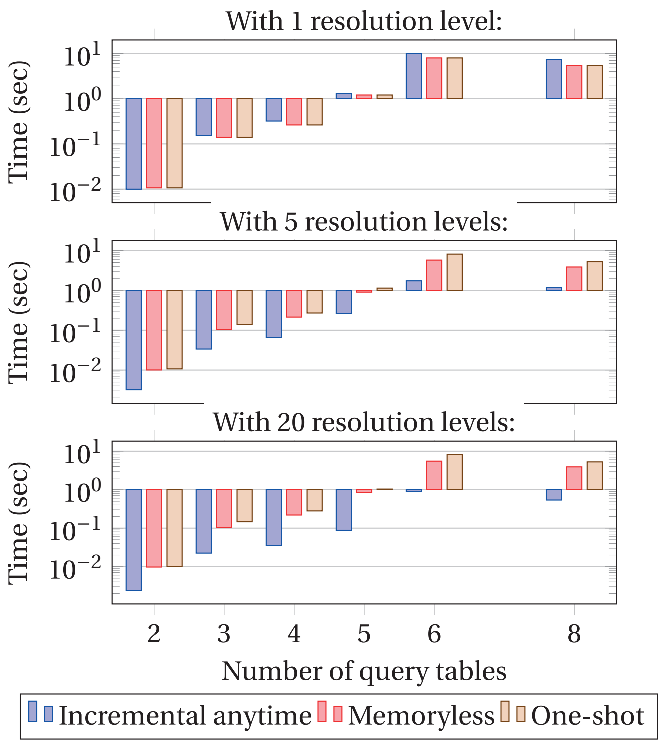


Figure 3.3 – Average time per optimizer invocation for TPC-H sub-queries and target precision $\alpha_T = 1.01$

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 MOQO, as demonstrated in the previous chapter. 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 3.4 shows analogous results for target precision $\alpha_T = 1.005$ (and $\alpha_S = 0.5$); using that

[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.

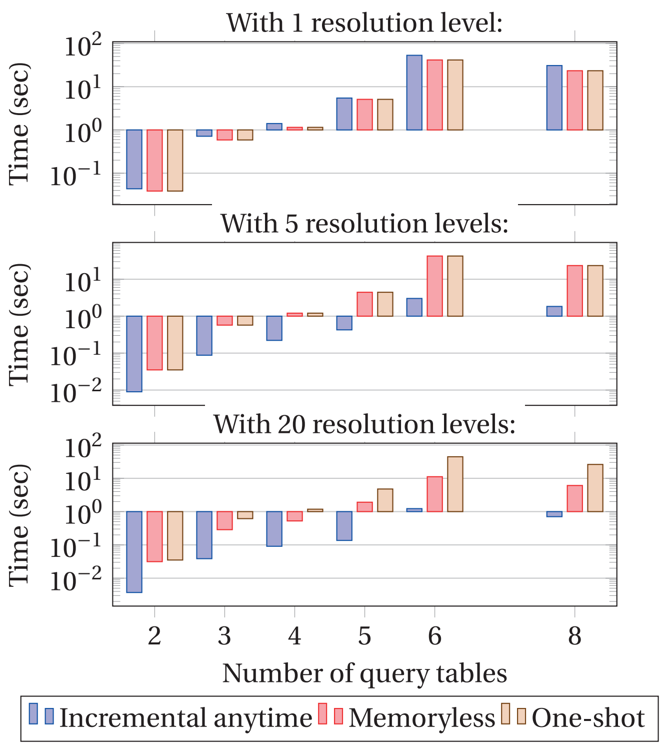


Figure 3.4 – Average time per optimizer invocation for TPC-H sub-queries and target precision $\alpha_T = 1.005$

target precision during pruning, all evaluated algorithms guarantee the generation of 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 3.5 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.

## 3.7 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.

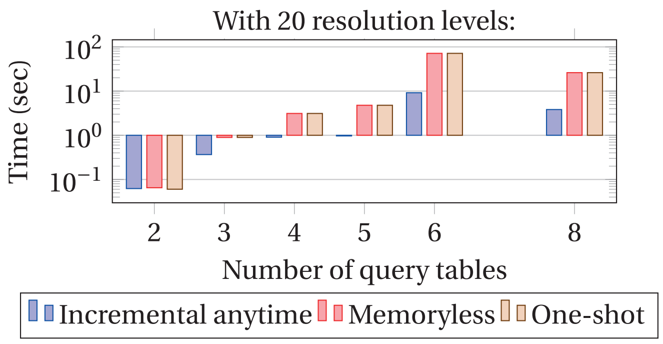


Figure 3.5 – Maximal time per optimizer invocation for TPC-H sub-queries and target precision $\alpha_T = 1.005$

# 4 Pre-Computation

On a high level, the algorithms presented in the last two chapters make query optimization with multiple cost metrics practical by significantly reducing optimization time compared to the exhaustive algorithm. An alternative way to make multi-objective query optimization practical, is to move optimization before run time. In that case, optimization may take a long time. Since it happens before run time, the constraints on optimization time are however relaxed. It is possible to move optimization before run time if the queries received at run time correspond to query templates that are known in advance. In this chapter, we will see an approach for pre-computing all Pareto-optimal plans for each possible instance of a given query template. This requires however to generalize the problem model of query optimization even further than in the previous chapters by considering at the same time multiple cost metrics and multiple parameters. Parameters represent unspecified parts in the query template. The resulting optimization problem, multi-objective parametric query optimization, generalizes many previously proposed problem variants in query optimization. We will see that multi-objective parametric query optimization differs in many aspects from other query optimization variants and needs to be solved by specialized optimization algorithms.

## 4.1 Introduction

Classical Query Optimization (CQ) models the cost of a query plan as a scalar cost value $c \in \mathbb{R}$. The optimization goal is to find the plan with minimal cost for a given query. Multi-Objective Query Optimization (MQ) [60, 88, 137] generalizes the classical model and associates each query plan with a cost vector $\mathbf{c} \in \mathbb{R}^n$ describing the cost of the plan according to multiple cost metrics. The optimization goal is to find a set of query plans that are all Pareto-optimal, meaning that no other plan has better cost according to all cost metrics at the same time. Parametric Query Optimization (PQ) [59, 73, 28] generalizes the classical model in a different way and associates each query plan with a cost function $c : \mathbb{R}^n \to \mathbb{R}$ describing the cost of the plan as function of multiple parameters whose values are not known at optimization time. The optimization goal is to find a plan set that contains an optimal plan for each possible combination of parameter values. In this chapter, we introduce Multi-Objective Parametric

Query Optimization (MPQ) and describe and analyze corresponding query optimization algorithms; MPQ generalizes and unifies the cost models of MQ and of PQ at the same time by representing the cost of a query plan as vector-valued function $\mathbf{c} : \mathbb{R}^n \rightarrow \mathbb{R}^m$. This allows to model multiple parameters as well as multiple cost metrics and is required in the following example scenarios.

**Example 7.** *A Cloud provider lets users query a large scientific data set over a Web interface. Query processing takes place in the Cloud. User queries correspond to query templates such as* `SELECT * FROM Table1 WHERE P1 AND P2` *where* `P1` *and* `P2` *represent unspecified predicates; users submit queries by specifying those predicates in the Web interface. Query processing time in the Cloud can often be reduced when accepting higher monetary fees [88]. After having submitted a query, users are therefore provided with a visualization of possible tradeoffs between execution time and monetary fees (that are realized by alternative query plans) and can select their preferred tadeoff. To speed up this process, the Cloud provider calculates all relevant query plans for each query template in a preprocessing step. The selectivities of the predicates are unknown at preprocessing time and must be represented as parameters, execution time and monetary fees are the two cost metrics. A query plan is relevant if there is at least one point in the parameter space for which its time-fees tradeoff is Pareto-optimal, meaning that no alternative plan has both, lower fees and lower execution time. Figure 4.1 illustrates the preprocessing result in this scenario (for a query with two unspecified predicates).*

**Example 8.** *Embedded SQL queries are a classical use case for PQ [73, 28]: to avoid query optimization overhead at run time, all potentially relevant query plans are calculated in advance for a given query template. Parameters model the selectivity of unspecified predicates or the amount of buffer space that is available at run time. Execution time is the only cost metric in the classical setting. In the context of approximate query processing [12], execution time can however be traded against result precision. In such a scenario, the two metrics execution time and result precision both must be considered during optimization. The optimal query plan is selected at run time based not only on concrete parameter values but also on a policy that determines the optimal tradeoff between result precision and execution time, based for instance on the current system load or on minimum precision requirements for one specific invocation.*

The kind of query optimization that is described in the example scenarios requires to consider multiple parameters and multiple cost metrics; this is a novel variant of query optimization that we call MPQ. Figure 4.2 describes the context of MPQ: MPQ takes place before run time; the input to MPQ is a query associated with parameters. A parameter may represent any quantity that influences the cost of query plans and is unknown at optimization time. The goal of MPQ is to generate a complete set of relevant plans, meaning a set that contains a plan $p^*$ for each possible plan $p$ and each point in the parameter space $\mathbf{x}$ such that $p^*$ has at most the same cost as $p$ at $\mathbf{x}$ according to each cost metric. Formulated differently, the goal is to find a set of Pareto-optimal query plans for all points in the parameter space. As in PQ [73], all relevant query plans are generated in advance so that no query optimization is required at run time.

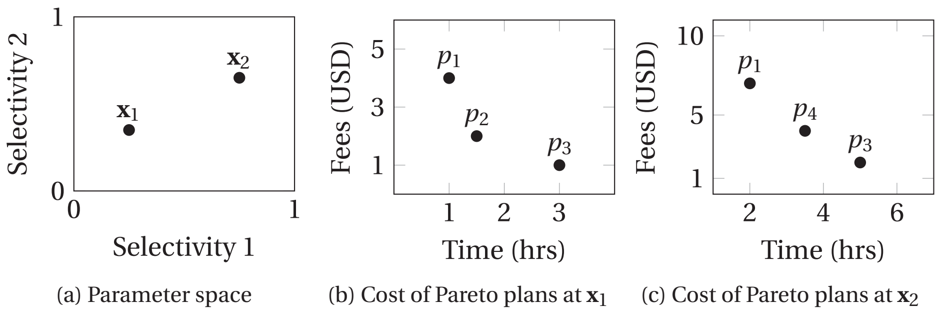


Figure 4.1 – MPQ associates each point $\mathbf{x}$ in the parameter space with a set of Pareto-optimal query plans $\{p_i\}$ (the illustration uses cost metrics and parameters from Scenario 1)

### 4.1.1 State-of-the-Art

MPQ is a generalization of MQ and of PQ; it is not possible to apply existing MQ or PQ algorithms to MPQ since PQ algorithms support only one cost metric and MQ algorithms do not support parameters. It may at first seem possible to model cost metrics as parameters; if all but one cost metric could be represented as parameters then PQ algorithms could be applied. Trying to model for instance monetary fees as a parameter in Scenario 7 (such that execution time becomes a function of predicate selectivities and monetary budget) leads however to the following problems: First, existing PQ algorithms [78, 59, 74, 18, 51, 32] usually assume that the value domain of each parameter is known in advance. This is realistic for predicate selectivity or the available amount of buffer space but not for monetary fees, as finding the minimal execution fees for a given query is a hard optimization problem all by itself. Second, cost metrics and parameters have different semantics: Assume for instance that alternative query plans for a given query have execution fees between 1 and 10 USD and that a plan $p$ priced at 5 USD has lower execution time than all plans with higher fees. The result set of MPQ should only contain $p$ but none of the more expensive plans since $p$ is always preferable over them. A PQ algorithm (e.g., [59, 73]) would however generate plans with minimal execution time for each possible cost value between 6 and 10 USD, as the goal in classical PQ is to cover the whole parameter space by optimal plans (while the goal in MPQ is not to cover the whole cost space). The result set of PQ can be larger than the result set of MPQ by an arbitrary factor and result set size relates to optimization time. Additional problems arise since parameter domains are usually assumed to be connected intervals while cost values may be sparsely distributed in the total cost range. Altogether, transforming a MPQ problem into a PQ problem by modeling cost metrics as parameters seems inappropriate. A popular branch of PQ algorithms decomposes a PQ problem into multiple non-parametric CQ problems; it is however impossible to analogously decompose a MPQ problem into multiple non-parametric MQ problems for reasons outlined in Section 4.4. More related work is discussed in Section 4.3.

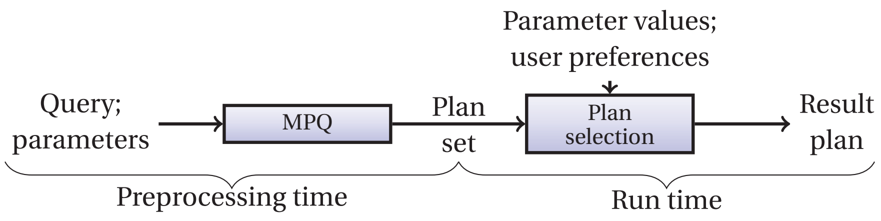


Figure 4.2 – The context of MPQ

### 4.1.2 Contribution and Outline

We summarize our contributions before providing details:

- We **formally analyze the** MPQ problem with piece-wise-linear (PWL) plan cost functions. We show in particular that the MPQ problem has no equivalent for certain fundamental properties of the PQ problem that have inspired the design of a broad class of PQ algorithms based on parameter space decomposition.
- We present **the first algorithms for MPQ**; those algorithms can deal with multiple cost metrics and parametric cost functions together. We present a generic MPQ algorithm that can deal with arbitrary plan cost functions and a specialization for PWL cost functions.
- We **formally analyze** our algorithms and show that both presented algorithms guarantee to generate all relevant query plans. We **experimentally evaluate** the algorithm for PWL cost functions in several example scenarios.

Section 4.2 introduces the formal model, Section 4.3 discusses related work. We analyze the MPQ problem in Section 4.4 and show that it differs from PQ in several important aspects. Section 4.5 presents and analyzes the Relevance Region Pruning Algorithm (RRPA). This is a generic algorithm for MPQ that can handle arbitrary plan cost functions. As many algorithms for CQ, MQ, and PQ, it is based on dynamic programming and generates and prunes query plans for joining table sets of increasing cardinality. The pruning function differs from prior approaches: Every query plan is associated with a region in the parameter space for which it is relevant (the Relevance Region, abbreviated RR). During pruning, this region is repeatedly reduced by comparisons with alternative plans. Plans are pruned once their RR becomes empty. We prove that RRPA formally guarantees to generate all relevant query plans for arbitrary queries.

The implementation of elementary RRPA operations such as adding cost functions and intersecting RRs depends on the considered class of cost functions. Most work on PQ focuses either on linear or on PWL cost functions which both can be stored and manipulated efficiently. Linear functions are however often a bad approximation for real plan cost functions [113] while PWL functions can approximate arbitrary cost functions up to an arbitrary degree of detail [73]. We therefore focus on PWL cost functions and present PWL-RRPA, a specialization of RRPA to PWL cost functions, in Section 4.6. We prove that all RRs that occur during the execution of PWL-RRPA belong to a limited class of shapes and propose data structures for representing cost functions and RRs. We provide pseudo-code for implementing all elementary operations of PWL-RRPA efficiently on those data structures and analyze the resulting complexity. PWL-RRPA was experimentally evaluated in multiple scenarios; the results are discussed in Section 4.7.

## 4.2 Definitions

Our notation is similar to the ones used in the previous chapters. Nevertheless, we introduce notation from scratch to make the current chapter self-contained.

A **query** is represented by a set $Q$ of tables that need to be joined. A **query plan** specifies the join order and the operators executing scan and join operations. The symbol $\mathbb{O}$ denotes the set of available operators. Let $p_1$ and $p_2$ be two query plans that join disjoint sets of tables and $o \in \mathbb{O}$ a join operator. The function $Combine(p_1, p_2, o)$ designates the query plan that joins the results of $p_1$ and $p_2$ using operator $o$. Plans $p_1$ and $p_2$ are called **sub-plans** of the resulting plan. The function $\mathbb{P}(Q)$ denotes the set of all possible plans for query $Q$. The execution **cost** of a query plan can depend on **parameters** whose exact values are not known at query optimization time. Parameters represent for instance predicate selectivities or the amount of available buffer space at query execution time. Parameter values for a fixed set of parameters are represented as a vector $\mathbf{x}$ (bold font distinguishes vectors from scalar values in the following). The **parameter space** $\mathbb{X}$ is the set of possible parameter vectors. Query plans are compared according to a set $\mathbb{M}$ of **cost metrics** for which analytic cost models are available. Let $p$ be a query plan and $\mathbf{x}$ a parameter vector. The cost function $\mathbf{c}(p,\mathbf{x})$ estimates the cost of plan $p$ under the circumstances described by parameter vector $\mathbf{x}$. The cost function yields a vector $\mathbf{c}$ that contains one value for each cost metric. Let $m \in \mathbb{M}$ be a cost metric, then $\mathbf{c}^m$ denotes the cost value for that metric. The notation $\mathbf{c}(p)$ designates the cost function for a constant plan $p$ such that $\mathbf{c}(p)(\mathbf{x}) := \mathbf{c}(p,\mathbf{x})$.

**Example 9.** *This example is based on Scenario 7. Consider a query template containing three predicates that are specified at run time. The selectivities of those three predicates are three parameters, the value domain of each parameter is the continuous interval* $[0,1]$. *The selectivities of all three predicates together can be described by a three-dimensional vector (for instance,* $\mathbf{x} = (0.1, 0.5, 0.2)$ *if the first predicate has selectivity 10%, the second predicate has selectivity 50%, and the third predicate has selectivity 20%). The parameter space containing all possible parameter vectors is the three-dimensional space* $\mathbb{X} = [0,1]^3 \subseteq \mathbb{R}^3$. *The cost of a fixed query plan depends on the selectivities of the predicates and is measured according to the two cost metrics execution time and monetary fees, therefore* $\mathbb{M} = \{time, fees\}$. *The value domain for each of the two cost metrics is the set* $\mathbb{R}_+ \subseteq \mathbb{R}$ *of non-negative real numbers. The cost function* $\mathbf{c}(p)$ *of a fixed plan* $p$ *therefore maps three-dimensional parameter vectors to two-dimensional cost vectors:* $\mathbf{c}(p) : \mathbb{X} \to \mathbb{R}_+^2$.

Plan quality metrics for which a higher value is better (e.g., result precision in Scenario 8) can always be transformed into cost metrics for which a lower value is better (e.g., replace result precision $\theta \in [0,1]$ by precision loss $1-\theta$). Let $p_1$ and $p_2$ be two query plans that produce the same result. Plan $p_1$ **dominates** plan $p_2$ in all points of the parameter space in which $p_1$ has at most the same cost as $p_2$ according to each cost metric. The function $Dom(p_1, p_2) \subseteq \mathbb{X}$ yields the parameter space region where $p_1$ dominates $p_2$:

$$Dom(p_1,p_2) = \{\mathbf{x} \in \mathbb{X} | \forall m \in \mathbb{M} : \mathbf{c}^m(p_1,\mathbf{x}) \leq \mathbf{c}^m(p_2,\mathbf{x})\}$$

The plans $p_1$ and $p_2$ mutually dominate each other in parameter space regions where they have equivalent cost. Plan $p_1$ **strictly dominates** plan $p_2$ in all points of the parameter space in which $p_1$ dominates $p_2$ without having equivalent cost. The function $StD(p_1,p_2) \subseteq Dom(p_1,p_2)$ yields the parameter space region where $p_1$ strictly dominates $p_2$:

$$StD(p_1,p_2) = Dom(p_1,p_2) \setminus \{\mathbf{x} \in \mathbb{X} | \mathbf{c}(p_1,\mathbf{x}) = \mathbf{c}(p_2,\mathbf{x})\}$$

A plan's region of optimality is in PQ the parameter space region where no alternative plan has lower cost [73]. The multi-objective analogue to the region of optimality is the **Pareto region**; the Pareto region $pReg(p) \subseteq \mathbb{X}$ of plan $p$ is the parameter space region where no alternative plan from $\mathbb{P}(Q)$ producing the same result as $p$ strictly dominates $p$:

$$pReg(p) = \mathbb{X} \setminus (\bigcup_{p^* \in \mathbb{P}(Q)} StD(p^*,p))$$

A parametric optimal set of plans is in PQ a plan set that contains at least one cost-optimal plan for each point in the parameter space [73]. The multi-objective analogue is a **Pareto plan set (PPS)**; $P \subseteq \mathbb{P}(Q)$ is a PPS iff it contains for each possible plan $p^* \in \mathbb{P}(Q)$ and each parameter vector $\mathbf{x} \in \mathbb{X}$ at least one plan plan that dominates $p^*$ for $\mathbf{x}$:

$$\forall p^* \in \mathbb{P}(Q)\ \forall \mathbf{x} \in \mathbb{X}\ \exists p \in P : \mathbf{x} \in Dom(p,p^*)$$

**Example 10.** *Let $p_1$, $p_2$, and $p_3$ be three plans for the same query. Assume there is only one parameter $\sigma \in [0,1]$ ($\mathbb{X} = [0,1]$) that represents the selectivity of an unknown predicate. The two cost metrics time and monetary fees are considered, therefore $\mathbb{M} = \{time, fees\}$. The plans have the following cost functions: $\mathbf{c}^{time}(p_1) = 2\sigma$, $\mathbf{c}^{fees}(p_1) = 3$, $\mathbf{c}^{time}(p_2) = 0.5+\sigma$, $\mathbf{c}^{fees}(p_2) = 2$, and plan $p_3$ has the same cost as $p_2$. The following relationships hold among others: Plans $p_2$ and $p_3$ mutually dominate each other in the entire parameter space: $Dom(p_2,p_3) = Dom(p_3,p_2) = [0,1]$. Plan $p_2$ strictly dominates $p_1$ for $\sigma > 0.5$. The Pareto region of $p_1$ is the selectivity interval $[0,0.5]$. The Pareto regions of $p_2$ and $p_3$ are the entire parameter space. The sets $\{p_1,p_2\}$ and $\{p_1,p_3\}$ both form a PPS.*

A **Pareto plan** designates in the following a plan in a PPS. A **relevance mapping (RM)** for a PPS $P$ maps each Pareto plan to a **relevance region (RR)** in the parameter space such that we can restrict our attention to the plans whose RR includes $\mathbf{x}$ whenever we need to find the best

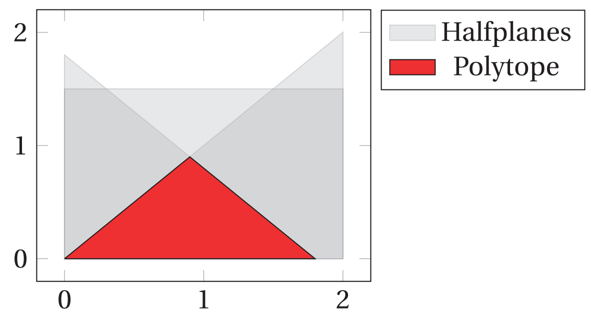


Figure 4.3 – Two-dimensional convex polytope as intersection of three halfplanes

plans for a parameter space point $\mathbf{x} \in \mathbb{X}$:

$$\forall p^* \in \mathbb{P}(Q)\, \forall \mathbf{x} \in \mathbb{X}\, \exists p \in P : \mathbf{x} \in relM(p) \cap Dom(p, p^*)$$

The RR of a plan can be different from its Pareto region. The algorithm presented in Section 4.5 uses RMs and discards plans with empty RRs. The **Multi-objective parametric query optimization (MPQ) problem** is the focus of this chapter. An MPQ problem is defined by a query $Q$, a parameter space $\mathbb{X}$, and a set of cost metrics $\mathbb{M}$. Any PPS for $Q$ is a solution to the MPQ problem.

We introduce a restricted variant of MPQ, the next definitions are prerequisites. An $m$-dimensional **convex polytope** is a set of points in $\mathbb{R}^m$ that *i)* is convex, meaning that any two points in the convex polytope are connected by a line segment that completely lies within the convex polytope again, and *ii)* corresponds to the intersection of a finite set of halfspaces, a halfspace being the set of solutions to a linear inequality of the form $\mathbf{w}^T \cdot \mathbf{x} \leq b$ with $\mathbf{w}, \mathbf{x} \in \mathbb{R}^m$ and $b \in \mathbb{R}$. Figure 4.3 illustrates how a convex polytope is constructed by intersecting three halfspaces in $\mathbb{R}^2$. The cost function $\mathbf{c}(p, \mathbf{x})$ of a plan $p$ is **linear** in the entire parameter space, if for each cost metric $m \in \mathbb{M}$, there is a weight vector $\mathbf{w}_m$ and a constant $b_m \in \mathbb{R}$ such that $\mathbf{c}(p, \mathbf{x}) = \mathbf{w}_m^T \cdot \mathbf{x} + b_m$ for each $\mathbf{x} \in \mathbb{X}$. The cost function is **piecewise-linear (PWL)** if the parameter space can be partitioned into convex polytopes such that $\mathbf{c}(p, \mathbf{x})$ is linear in each polytope. Note that PWL cost functions may have discontinuities between regions in which they are linear. PWL functions are of high practical relevance since they can approximate arbitrary functions [74]. Most work on PQ (e.g., [59, 73]) restricts the PQ problem by assuming either linear or PWL cost functions. In analogy to that, we introduce a restricted variant of the MPQ problem: **PWL-MPQ** assumes that all vector-valued cost functions are PWL and that the parameter space itself forms a convex polytope (which is a standard assumption in PQ [73]). The PWL-MPQ problem is analyzed in Section 4.4 and a corresponding optimization algorithm is presented in Section 4.6. This algorithm exploits that the parameter space in PWL-MPQ can be partitioned into **linear regions** for a plan set $P$: a linear region is a convex polytope in the parameter space for which all plans in $P$ have linear cost functions.

## 4.3 Related Work

We introduced four different variants of query optimization in Section 4.1.1 (CQ, PQ, MQ, and MPQ) and justified why existing algorithms cannot be applied for MPQ. We discuss related work in PQ and MQ in more detail now.

**PQ algorithms** associate query plans with cost functions instead of cost values. The cost functions depend on parameters that represent for instance predicate selectivities. The goal in PQ is usually to generate a plan set that contains one optimal plan for each possible parameter value combination [59, 73, 74, 28]. Many approaches to PQ are based on parameter space decomposition [59, 73, 74, 51, 28]. They repeatedly invoke a standard optimizer to generate optimal plans for fixed parameter values (if the parameter values are fixed then the cost of a query plan can be modeled as a constant value again) in order to decompose the parameter space into regions in which a single plan is optimal. We will see in Section 4.4 why similar approaches fail for MPQ. Another branch of PQ algorithms [62, 43, 73, 74, 18, 32] is based on dynamic programming, similar to the CQ algorithm by Selinger [116]. They are specific to PQ since they consider only one cost metric during pruning (some approaches consider robustness in addition to execution time [17, 10] but robustness is directly derived from execution time and not an independent cost metric) and use data structures and corresponding manipulation functions that are intrinsically specific to assumptions that hold in PQ but not in MPQ (e.g., many PQ algorithms model the parameter space region in which a plan is optimal as convex polytope which works for PQ with PWL cost functions but not for MPQ with PWL cost functions as shown in Section 4.4). Using PQ algorithms for MPQ would require that the optimal plan according to one cost metric is always guaranteed to be optimal according to all other cost metrics. This case is unrealistic; even more so since many relevant cost metrics are anti-correlated (e.g., result precision and processing time in approximate query processing [12]). Ioannidis et al. [78] use randomized algorithms for PQ; they do not support multiple cost metrics. Randomized algorithms can never offer formal worst-case guarantees on generating complete plan sets, unlike the algorithms presented in this chapter. Classical PQ deals with unknown parameter values by generating all plans that could be relevant. Other approaches define probability distributions over parameter values with the goal to generate one robust plan [17, 10] or one plan that minimizes expected cost [40]. In contrast to that, classical PQ aims at scenarios where new information becomes available at run time that should be considered during plan selection.

**MQ algorithms** compare query plans according to several cost metrics. The goal is to find a plan that represents the best compromise between conflicting metrics according to user preferences. The single-objective query optimization algorithm by Selinger has been generalized to MQ [60, 137]: plans producing the same result are compared according to multiple cost metrics during pruning and plans that are not Pareto-optimal are discarded. The latter approach can deal with a broad range of cost metrics but does not support parameters. Other MQ algorithms are tailored to specific combinations of cost metrics and user preference functions that allow efficient pruning [81, 147, 11, 12]. They allow for instance only cost metrics

| **Case of Single Cost Metric** | **Case of Multiple Cost Metrics** |
|---|---|
| **(S1)** If the same plan is optimal for two points in a linear parameter space region, then that plan is also optimal on the line connecting those two points. | **(M1)** If the same plan is Pareto-optimal for two points in a linear parameter space region, then this plan is **not** necessarily Pareto-optimal on the line connecting those two points. |
| **(S2)** Each plan has one connected region within a linear parameter space region for which it is optimal. This region is either empty or forms a convex polytope. | **(M2)** The Pareto region of a plan within a linear region is **not** necessarily connected and the connected parts of it do **not** form convex polytopes in general. |
| **(S3)** If the same plan is optimal for all vertices of a convex polytope in a linear parameter space region, then that plan is optimal for all points within the polytope. | **(M3)** If all vertices of a convex polytope in a linear parameter space region have the same set of Pareto plans, then **(M3a)** those plans are **not** necessarily Pareto-optimal for all points of the polytope, and **(M3b)** plans can be Pareto-optimal within the polytope that are not Pareto-optimal on the vertices. |

Table 4.1 – Comparing the case of one cost metric and the case of multiple cost metrics in parametric query optimization; all statements refer to linear regions in the parameter space

for which the cost of a query plan is calculated as weighted sum over the cost of its sub-plans [147]; this is however not possible in many relevant scenarios (e.g., the execution time of a plan equals the maximum over the execution times of its sub-plans if they are executed in parallel). None of those approaches supports parameters. The algorithms that we present in this chapter place only minimal restrictions on the cost metrics (see Section 4.5.2) and allow parameters which is required to solve MPQ problems. Yet another branch of MQ algorithms separate multi-objective optimization from join ordering; they produce for instance a time-optimal join tree first and configure operators within that tree considering multiple cost metrics later [61, 107]. Such approaches are not applicable to MPQ since it is unrealistic to find one join tree that is optimal for all parameter values (parameters such as predicate selectivities clearly have strong influence on the optimal join order). Algorithms for multi-objective data flow optimization [124, 125, 88] are not applicable to query optimization with join reordering.

## 4.4 Problem Analysis

We analyze the newly introduced MPQ problem. The PQ problem (i.e., the MPQ problem with only one cost metric) was already analyzed in prior work [59]. The MPQ problem is a generalization of the PQ problem and the following analysis therefore focuses on pointing out differences between the PQ problem and the MPQ problem. We will see in Section 4.4.1 that having multiple cost metrics instead of only one changes many fundamental problem properties. This has important implications on the design of MPQ algorithms that we discuss in Section 4.4.2.

### 4.4.1 Analysis

Most work on PQ assumes that all cost functions are PWL [59, 73, 51]. We make the same assumption in the following. Our comparison between PQ and MPQ focuses on three problem properties that have been shown to hold for PQ. Those three problem properties were already called the *guiding principles of PQ* [51] since many PQ algorithms exploit them in one way or another [59, 73, 51], assuming that they hold either over the whole parameter space [59, 73] or at least locally [51]. We will see that the guiding principles do not hold anymore for MPQ which makes many successful approaches to PQ inapplicable to MPQ. Table 4.1 summarizes the differences between PQ and MPQ. The left column contains statements about PQ that were proven by Ganguly [59]; the right column contains the adapted statements for MPQ that are proven next. All statements refer to linear regions (convex polytopes in the parameter space in which all compared cost functions are linear for each cost metric).

**Theorem 14.** *The parameter space can be partitioned into linear regions for an arbitrary set of cost functions.*

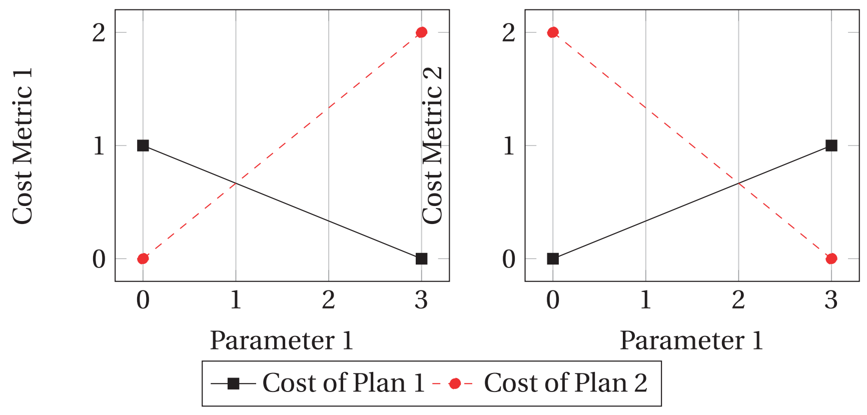
*Proof.* Given only one cost metric, the parameter space can always be partitioned into linear regions according to results from PQ [73]. Denote by $C_i$ the partitioning according to the $i$-th cost metric for $1 \leq i \leq M$ (represented as a set of polytopes). Then $\{c = c_1 \cap \ldots \cap c_M | c_i \in C_i\}$ is a partitioning of the parameter space into linear regions according to all cost metrics. The partitions are intersections of convex polytopes and therefore convex polytopes themselves. □

We refer to the three statements about PQ by **S1**, **S2**, and **S3** in the following, and to the three statements about MPQ by **M1**, **M2**, and **M3**.

**Theorem 15.** *Let $p_1$ and $p_2$ two arbitrary plans and $X \subseteq \mathbb{X}$ a linear region for $\{p_1, p_2\}$. Then the region $D$ within $X$ in which $p_1$ dominates $p_2$ forms a convex polytope.*

*Proof.* Denote by $D_m \subseteq X$ the region in which $p_1$ is better or equivalent to $p_2$ according to cost metric $m \in \mathbb{M}$. Each region $D_m$ forms a convex polytope (see results on PQ with linear cost functions [59]). Plan $p_1$ dominates $p_2$ in the region in which it is better or equivalent to $p_2$ according to all cost metrics. Region $D$ corresponds therefore to the intersection of the $D_m$: $D = \cap_{m \in \mathbb{M}} D_m$. A convex polytope is an intersection of halfplanes. Therefore, the intersection of convex polytopes is a convex polytope again. □

The following series of counter-examples proves the statements from Table 4.1. The multi-objective equivalent of an optimal plan is a Pareto-optimal plan. Statement **S1** about PQ does not generalize to the multi-objective case. Figure 4.4 shows a corresponding counter-example.



| Parameter Range | Pareto Plans |
|---|---|
| $[0,1)$ | Plan 1, Plan 2 |
| $[1,2)$ | Plan 1 |
| $[2,3]$ | Plan 1, Plan 2 |

Figure 4.4 – If a plan is Pareto-optimal for two parameter values, it is not necessarily Pareto-optimal for the values in between

The example shows the two-dimensional cost function of two plans within a one-dimensional parameter space. Plan 1 is Pareto-optimal in the whole parameter space (parameter value range $[0,3]$). Plan 2 is however only Pareto-optimal for the parameter value ranges $[0,1)$ and $[2,3]$ but not for parameter values between 1 and 2. The example is minimal for MPQ since having less than two cost metrics leads to PQ and having less than one parameter leads to MQ. The negative result therefore applies to MPQ in general.

This example shows at the same time that Pareto regions are not necessarily connected (first part of **M2**). Figure 4.5 illustrates the second part of statement **M2**: the connected parts of the Pareto region are not necessarily convex. The example depicted in Figure 4.5 uses two plans and a two-dimensional parameter space. The example requires a two-dimensional parameter space since connected regions in a one-dimensional parameter space always form convex polytopes. Let $\mathbf{c}_1(x_1,x_2) = (x_1,x_2)$ be the two-dimensional cost function of plan 1 (the two-dimensional identity function) and $\mathbf{c}_2(x_1,x_2) = (1,1)$ the cost function of plan 2. The region in which plan 1 dominates plan 2 forms a convex polytope as depicted in Figure 4.5. The remaining region is the Pareto region of plan 2. Figure 4.5 shows clearly that the Pareto region is not convex.

The example in Figure 4.4 also proves **M3a**. The example in Figure 4.6 proves **M3b**. Figure 4.6 shows cost functions of three plans for two cost metrics and one parameter. Plan 3 is Pareto-optimal for the parameter range $(0.5,1.5)$ but neither for the range $[0,0.5]$ nor for the range $[1.5,2]$. The cost functions in our examples are not monotone but the examples can be adapted (just turn the figures counterclockwise by 45 degrees). A common assumption in PQ is that

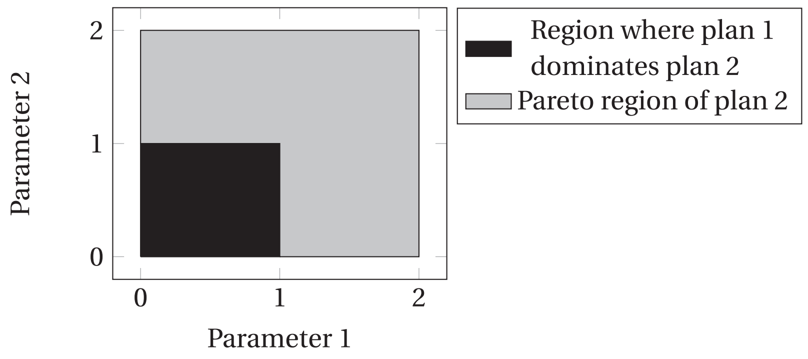


Figure 4.5 – Pareto regions are not necessarily convex

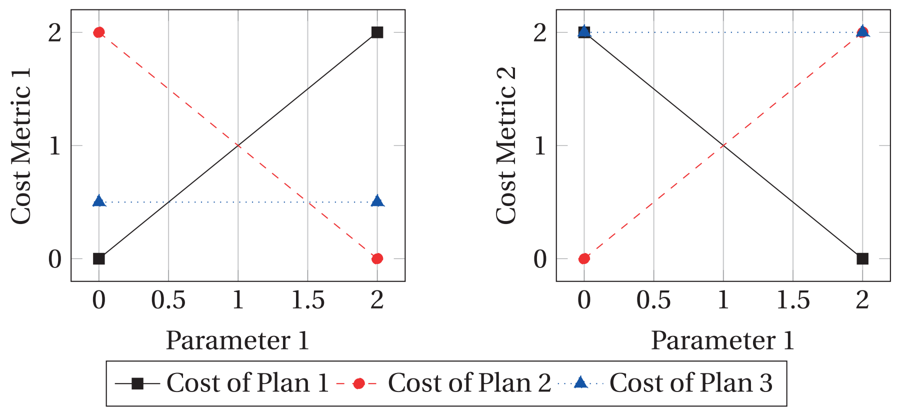
plan cost functions are monotone in the parameters [17]. We see that this assumption does not change our negative results.

### 4.4.2 Implications on Algorithm Design

The three properties of the PQ problem that are listed in the left column of Table 4.1 have allowed to design PQ algorithms that split one PQ problem into several CQ problems. This approach has the advantage that an existing query optimizer for CQ can be turned into an optimizer for PQ with relatively low implementation overhead: the code of the existing CQ optimizer remains mostly unchanged (this is why such approaches to PQ are called *non-intrusive* [73]) and only a relatively small piece of code has to be added that splits the PQ problem into several CQ problems. We will see now, why such approaches fail for MPQ.

The Recursive Decomposition Algorithm proposed by Hulgeri and Sudarshan [73] is a non-intrusive PQ algorithm and works as follows: Given a convex polytope in the parameter space, the algorithm calculates an optimal plan for each vertex of that polytope (using a CQ query optimizer). If the same plan is optimal for each vertex, then that plan is optimal for every point within the polytope (according to statement **S3** from Table 4.1) and no further decomposition is necessary. If different plans are optimal for different vertices, then the polytope is decomposed into fragments and the algorithm is recursively applied to each fragment.

The described algorithm is representative for other non-intrusive approaches to PQ [59, 73, 74] since all of them successively decompose the parameter space into fragments in which only one plan is optimal. Statement **S3** is crucial for all those algorithms since it leads to a sufficient condition for checking whether further decomposition is unnecessary. Statement **M3** shows that no analogue condition can be found for MPQ: even if the same set of plans is Pareto-optimal for all vertices of a convex polytope in the parameter space, it may still be necessary to decompose that polytope further in order to find all Pareto plans (according to Statement **M3b**). This means that it is not possible to generalize non-intrusive algorithms for PQ to MPQ (which would allow to split one MPQ problem into several MQ problems to which existing MQ algorithms could be applied [60]). Motivated by this insight, we propose quite a different



| Parameter Range | Pareto Plans |
|---|---|
| $[0, 0.5]$ | Plan 1, Plan 2 |
| $(0.5, 1.5)$ | Plan 1, Plan 2, Plan 3 |
| $[1.5, 2]$ | Plan 1, Plan 2 |

Figure 4.6 – If a plan is not Pareto-optimal for two parameter values, it can still be Pareto-optimal for the values in between

approach to MPQ in the following section.

## 4.5 Generic Algorithm

In this section, we present the Relevance Region Pruning Algorithm (RRPA) for MPQ. The algorithm associates each query plan with a RR in the parameter space that is used during pruning to detect irrelevant plans. The algorithm is generic and not specific to PWL cost functions. Section 4.5.1 describes the algorithm and Section 4.5.2 proves that RRPA finds complete PPSs for arbitrary MPQ problem instances. We do not explicitly describe how to deal with nested queries during optimization; techniques for decomposing complex SQL statements into simple SPJ query blocks have been proposed in prior work [116].

### 4.5.1 Outline of Algorithm

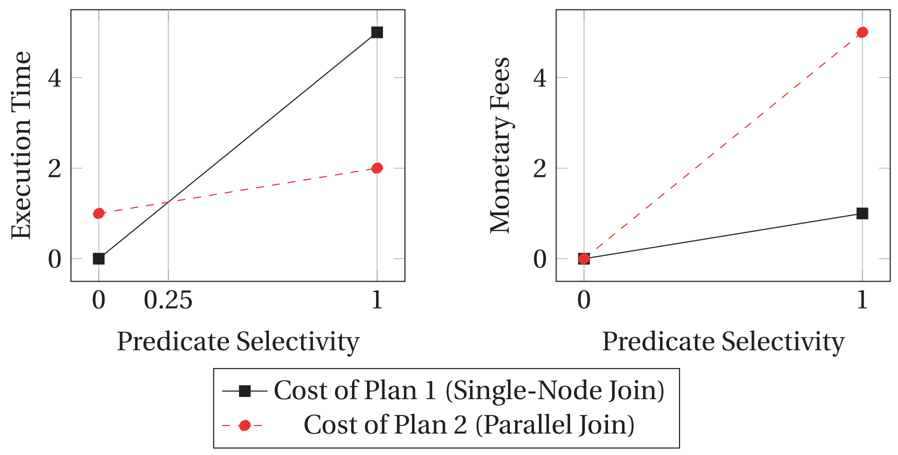
The analysis from the previous section has shown that trying to adapt non-intrusive PQ algorithms to MPQ is not a promising direction. We adopt a dynamic programming (DP) based approach instead, calculating optimal plans for joining table sets out of optimal plans for joining subsets. Such an approach seems promising because DP has been widely used for designing algorithms in CQ [116], MQ [60], and PQ [73]. Algorithm 7 shows pseudo-code of RRPA. The main function takes a query $Q$ as input and returns a PPS for $Q$. The algorithm uses two families of global variables: For each sub-query $q \subseteq Q$, variable $\mathscr{P}^q$ will eventually contain a PPS for $q$ and variable $\mathscr{R}^q$ a corresponding RM (let $p \in \mathscr{P}^q$ a plan for $q$, then $\mathscr{R}^q(p)$

designates the RR of $p$). We assume that the plan sets are initially empty. RRPA first calculates PPSs and RMs for each base table $q \in Q$; it considers all possible scan plans for each base table and prunes out plans that are dominated in the entire parameter space. Details of the pruning function are discussed later. After the base tables, RRPA treats table sets in ascending order of cardinality. An auxiliary function generates the PPS for joining a table set $q \subseteq Q$ by considering all possible splits of $q$ into two non-empty subsets (each split represents one specific pair of operands for the last join), all possible operators for the last join, and all pairs of plans for generating the inputs to the last join (those plans are selected out of the PPSs that were calculated before). A tentative plan is generated for every combination of operands, operator, and sub-plans. This plan is compared pairwise against all other plans that generate the same result and are already contained in $\mathscr{P}^q$. Those comparisons happen in the pruning function. The goal is to identify and discard suboptimal plans that are not required to form a PPS.

Pruning is based on the concept of RRs that was introduced in Section 4.2. Every plan is associated with a RR in the parameter space for which no alternative plan is known that has equivalent or dominant cost. The RR of a newly generated plan is initialized by the full parameter space. It is reduced during a series of comparisons between the newly generated plan and the old plans joining the same tables. At every comparison, the RR of the new plan is reduced by the points in the parameter space for which an old plan dominates the new plan. If the RR of the new plan becomes empty, it is discarded. Otherwise, the new plan is inserted. Before inserting the new plan, the RRs of the old plans are reduced by regions in which they are dominated by the new plan. Old plans with empty RRs are discarded. The following example illustrates the pruning method.

**Example 11.** *We revisit Scenario 7. Figure 4.7 shows the cost functions of two query plans that join the same two tables. The amount of data that needs to be joined depends linearly on the selectivity of one predicate; all cost functions therefore depend on this parameter. Plan 1 uses a single-node join while plan 2 uses a parallel join involving multiple nodes. Plan 1 executes faster than plan 2 for small amounts of input data since no data needs to be shuffled around in the network (assuming that all required input data resides initially on one node). Plan 2 executes faster for larger amounts of input data due to parallelization. The monetary fees of plan 2 are however always higher than the fees for plan 1, since the fees are proportional to the total work (summing up over different nodes) and the total amount of work increases by parallelization.*

*Assume that plan 1 was generated before plan 2. The RR of plan 2 directly after its creation is the entire parameter space* $[0,1]$. *Plan 2 is pruned with all previously generated plans for joining the same tables, this is only plan 1 in our example. Plan 1 is preferable over plan 2 according to execution time and monetary fees at the same time as long as the selectivity is smaller than* $0.25$. *The RR of plan 2 is therefore reduced by the interval* $[0,0.25]$ *such that plan 2 remains relevant for the interval* $[0.25,1]$. *Note that this example uses only linear cost functions that depend on only one parameter while* RRPA *can work with arbitrary cost functions that depend on an arbitrary number of parameters.*



| Optimization Phase | Relevance Region of plan 2 |
|---|---|
| After creating plan 2 | $[0,1]$ |
| After pruning plan 2 with plan 1 | $[0.25,1]$ |

Figure 4.7 – Illustration of pruning function

Algorithm 7 does not specify how elementary operations such as adding cost functions or intersecting relevance regions are implemented. The best way of implementing those operations depends on the considered class of cost functions (which also implicitly determines the class of RR shapes that one needs to consider). It is therefore impossible to specify an implementation for the generic case. For the same reason it is not possible to analyze the time complexity of RRPA. We will however present a specialized version of RRPA for PWL cost functions in Section 4.6 and analyze its complexity.

### 4.5.2 Proof of Completeness

We prove that RRPA generates complete PPSs for arbitrary input queries. We make the common assumption that the *Principle of Optimality* (POO) [60] holds for each cost metric: replacing a sub-plan $p_S$ within a query plan $p$ by an alternative sub-plan $p'_S$ that has better or equivalent cost than $p_S$ for a specific parameter vector $\mathbf{x}$ and according to a specific cost metric $m$, can only lead to a plan whose cost according to $m$ is better than or equivalent to the one of $p$ for $\mathbf{x}$. The POO restricts the cost function of a plan with regards to the cost functions of its sub-plans but it does not restrict the shapes of cost functions in general.

The proof that RRPA generates PPSs is an induction over the number of tables to join. The following lemma will be used for the inductive step.

**Lemma 10.** *If* RRPA *generates PPSs and corresponding RMs for all queries that join up to $N$ tables then it also generates PPSs and corresponding RMs for queries that join up to $N+1$ tables.*

*Proof.* Let $Q$ be a query joining $N+1$ tables ($|Q| = N+1$), vector $\mathbf{x} \subseteq \mathbb{X}$ an arbitrary parameter vector, and $p$ an arbitrary plan for $Q$. Plan $p$ has two sub-plans, $p_1$ and $p_2$, that join at most $N$ tables each. Therefore, RRPA generates a plan $p_1^*$ that produces the same result as $p_1$ and dominates $p_1$ for $\mathbf{x}$. Additionally, $\mathbf{x}$ is included in the RR of $p_1^*$. RRPA also generates a plan $p_2^*$ with the analogous properties relative to $p_2$. The plans $p_1^*$ and $p_2^*$ can be combined into a plan $p^*$ that produces the same result as $p$ and dominates $p$ for $\mathbf{x}$ (due to the POO).

RRPA will generate $p^*$ and initialize its RR with the full parameter space. Plan $p^*$ is only pruned once its RR becomes empty during the pairwise comparisons with other plans. This can only happen, if RRPA keeps another plan that dominates $p^*$ for $\mathbf{x}$ and $\mathbf{x}$ will be included in that plan's RR. RRPA generates a PPS for query $Q$ and the corresponding RM since $p$ and $\mathbf{x}$ were chosen arbitrarily. □

The following theorem is the main result of this subsection.

**Theorem 16.** RRPA *generates PPSs for arbitrary MPQ problem instances.*

*Sketch.* The proof is an induction over the number of tables to join. Under the assumption that RRPA generates PPSs and corresponding RMs for single tables (the induction start), it also generates PPSs and corresponding RMs for arbitrary table sets according to Lemma 10 (the induction step). RRPA considers all possible plans for each base table and only discards plans that are dominated in the entire parameter space. This proves the induction start. □

## 4.6 Algorithm for Piecewise-Linear Cost Functions

RRPA presented in the last section is generic since it can deal with arbitrary cost functions. The pseudo-code of RRPA (Algorithm 7) left certain questions open such as how to represent RRs and how to efficiently intersect and reduce them; the answers to those questions depend on the considered class of cost functions. In this section, we present a specialized version of RRPA for PWL cost functions: PWL-RRPA. We propose data structures to represent cost functions and RRs in Section 4.6.1 and show how elementary operations can be efficiently implemented on them in Section 4.6.2. We show in Section 4.6.3 how the representation of parameter space regions can be simplified. In Section 4.6.4, we analyze the complexity of PWL-RRPA. Note that PWL-RRPA guarantees to generate PPSs for arbitrary PWL-MPQ problem instances as it is a specialization of RRPA.

### 4.6.1 Data Structures

Expressions of the form $\mathscr{R}^q(p)$ designate in Algorithm 7 the RR of a plan $p$ joining a table set $q$. Figure 4.8 describes the internal representation of RRs as entity-relationship diagram. A RR is represented by a set of convex polytopes, called the cutouts, such that a parameter space

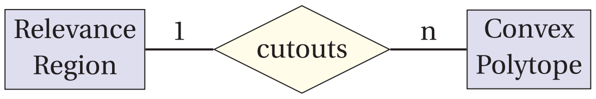


Figure 4.8 – Representation of relevance regions if all cost functions are piecewise-linear

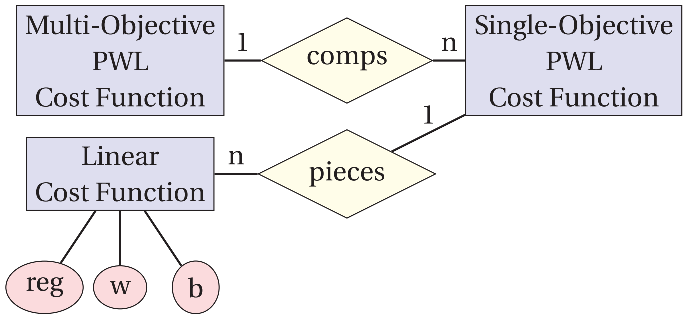

Multi-Objective PWL Cost Function
1
comps
n
Single-Objective PWL Cost Function
1
Linear Cost Function
n
pieces
reg
w
b

Figure 4.9 – Representation of multi-objective piecewise-linear cost functions

vector is contained in a RR if it is not contained in any of the cutouts. The following theorem justifies this representation.

**Theorem 17.** *Any relevance region that occurs during the execution of* PWL-RRPA *can be represented as complement of a set of convex polytopes.*

*Proof.* The RR of a new plan is the entire parameter space and can therefore be represented as the complement of an empty set. After initialization, the RR can get reduced several times by regions in which a plan is dominated by another. When comparing two plans with PWL cost functions, the parameter space can be partitioned into linear regions according to Theorem 14. The region in which one plan dominates another within a linear region forms a convex polytope according to Theorem 15. Therefore, the RR can still be represented as complement of convex polytopes after reduction. □

The cost function of a plan $p$ is represented by the expression $\mathbf{c}(p)$ in Algorithm 7. Figure 4.9 shows the internal representation of cost functions as entity-rela- tionship diagram. A multi-objective PWL cost function is composed out of one single-objective PWL cost function per cost metric. The PWL cost function is linear within parameter space regions that form convex polytopes. Each PWL function is therefore represented as a set of linear functions; each linear function is characterized by the parameter space region to which it applies (attribute $reg$ in Figure 4.9) and a weight vector (attribute $\mathbf{w}$ in Figure 4.9) with one weight per parameter together with the scalar base cost ($b$ in Figure 4.9) that define the linear function. The parameter space regions of the linear pieces must not overlap; then the PWL function can be evaluated for a specific parameter vector $\mathbf{x}$ by identifying the unique piece whose region contains $\mathbf{x}$ and evaluating the formula $b+\mathbf{w}^T\cdot\mathbf{x}$ to obtain the cost value. A multi-objective PWL function is evaluated by evaluating all its components according to the aforementioned method.

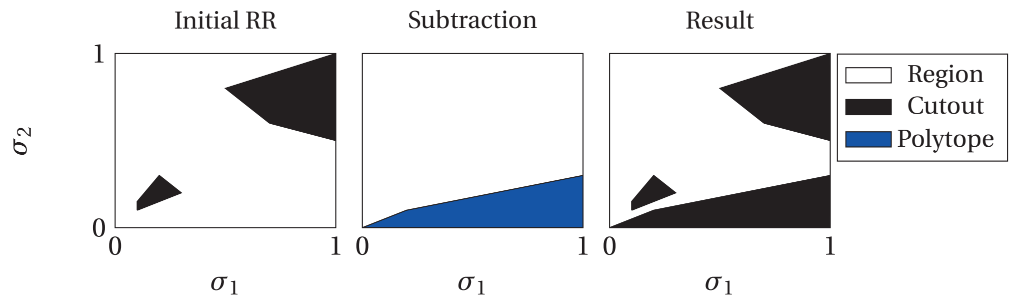


Figure 4.10 – Polytopes are subtracted from a relevance region by adding them as cutouts

PWL cost functions can approximate the real cost functions of single scan and join operations up to an arbitrary precision [73]. The accumulated cost of an entire query plan (using standard accumulation function such as minimum, maximum, and weighted sum) can therefore be represented as PWL function again; this fact has been used by prior PQ algorithms [73]. Generalizing this reasoning to the multi-objective case is trivial. Therefore, the representation proposed in Figure 4.9 covers each cost function that occurs during the execution of PWL-RRPA (assuming that the cost of single operations is approximated by PWL functions).

### 4.6.2 Implementation of Elementary Operations

PWL-RRPA performs two operations on RRs: it reduces the RR of a plan by the region in which it is dominated by another (e.g., Algorithm 7, Line 39) and it checks whether a RR is empty (Algorithm 7, Line 41). Algorithm 8 shows pseudo-code for both operations. The field specifier $.cutouts$ refers to Figure 4.8 and denotes the set of cutouts for a variable representing a RR. Convex polytopes are subtracted from a RR by adding them as cutouts, as illustrated in Figure 4.10.

Function IsEmpty is based on the following theorem.

**Theorem 18.** *A relevance region is empty iff the union of its cutouts forms a convex polytope that covers the entire parameter space.*

*Proof.* Let $C_i \subseteq \mathbb{X}$ be the set of cutouts. The RR is empty iff $\forall \mathbf{x} \in \mathbb{X} \exists i : \mathbf{x} \in C_i$. This is the case iff $\mathbb{X} \subseteq \cup_i C_i$ which is equivalent to $\mathbb{X} = \cup_i C_i$ since all cutouts are contained within the parameter space $\mathbb{X}$. As $\mathbb{X}$ forms a convex polytope according to the definition of the PWL-MPQ problem (see Section 4.2), the union of the cutouts of an empty RR is a convex polytope. □

The union of the cutouts may not be convex and may not form a polytope. Checking whether a region of arbitrary shape (the union of the cutouts) contains the parameter space is inefficient. It is therefore crucial to note that the containment check is only necessary in the special case that the union of cutouts forms a convex polytope. The algorithm by Bemporad et al. [22] checks whether a union of convex polytopes is a convex polytope again and constructs the

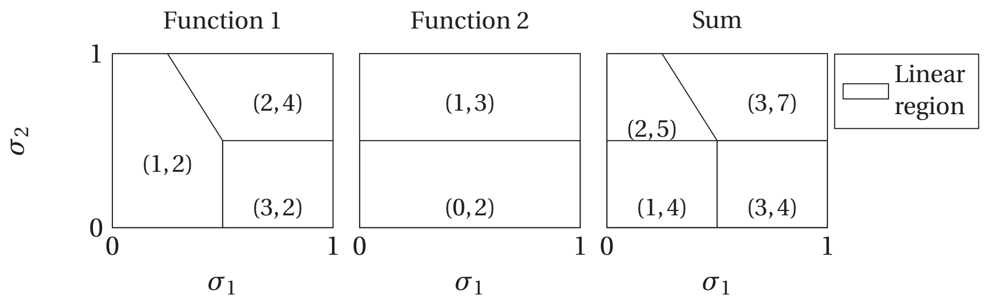


Figure 4.11 – To add single-objective cost functions, their weight vectors are added in each linear region

corresponding polytope in that case. Checking containment between two convex polytopes is a standard problem [80].

PWL-RRPA performs two operations on cost functions: It calculates the cost function of a new plan by accumulating the cost of its sub-plans (Algorithm 7, Line 26) and—given two cost functions—it calculates the region in which one dominates the other (e.g, Algorithm 7, Line 39). Algorithm 9 shows pseudo-code for both operations. The $comps$ relationship (see Figure 4.9) associates a multi-objective cost function with one single-objective function for each cost metric. We treat the $comps$ relationship as an array and refer to the single-objective cost function for metric $m$ by the notation $.[m]$. The function ACCUMULATECOST accumulates the cost of a new plan out of the cost of its sub-plans. It iterates over all cost metrics and calculates the cost function for each metric separately. For each metric, it partitions the parameter space into regions in which both sub-plans have linear cost functions. Each nonempty linear region becomes a piece in the cost function of the new plan. The weight vector of the new piece corresponds to the component-wise sum of the weight vectors of the two sub-plans and the join cost vector (denoted by $o.\mathbf{w}$ in the pseudo-code) in the corresponding parameter space region[1]; Figure 4.11 illustrates this step for a two-dimensional parameter space with parameters $\sigma_1$ and $\sigma_2$, the two-dimensional weight vectors are shown at the interior of their linear regions. The base cost of the new piece is the sum over the join base cost ($o.b$) and the base costs of the sub-plans. Cost is therefore accumulated by adding the cost of the sub-plans. The function trivially generalizes to scenarios where cost is accumulated as weighted sum, minimum, or maximum of two cost functions.

Function DOM returns a set of convex polytopes representing the region in which plan $p_1$ dominates plan $p_2$. A plan dominates another in regions in which it has better or equivalent cost according to each cost metric. Function DOM initially calculates for each cost metric $m$ the set $DomPolys_m$ of convex polytopes in the parameter space in which $p_1$ is better than or equivalent to $p_2$ according to $m$. In a second step, the function intersects the polytope sets

[1]To simplify the pseudo-code, we made the strong assumption that the cost function of the final join is always linear in parameter space regions in which the cost functions of the two sub-plans are linear. This is not true in general but the code can easily be generalized by first accumulating the cost of the sub-plans, and then accumulating the resulting cost and the join cost in a second step.

associated with specific cost metrics to obtain the region in which $p_1$ is better or equivalent according to all metrics.

### 4.6.3 Simplifying Parameter Space Regions

It is crucial to keep the representation of parameter space regions as simple as possible. For convex polytopes, this means that we want to represent them using as few constraints as possible. For RRs, it means that we want to represent them using the smallest possible number of cutouts. Our algorithm regularly tries to simplify the representation of parameter space regions. We found simplification steps to be indispensable for efficient optimization: optimization time decreases by two orders of magnitude when implementing the simplifications that are described in the following.

Algorithm 10 shows pseudo-code for the methods that we use to simplify parameter space regions. Convex polytopes are the basic components of all shapes that we represent. A convex polytope is described by a set of linear constraints. We can construct polytopes step by step by adding one constraint after the other one. Function ADDCONSTRAINTSIMP can be used to construct polytopes step by step. At each invocation, this function adds a constraint and tries to simplify the representation by removing redundant constraints. The auxiliary function CANREMOVE is used to verify whether a specific constraint can be removed. This function obtains as input a polytope, $poly$, and a constraint of that polytope, $C$. We use the notation $poly.constr$ to access the constraints that define a polytope. Function CANREMOVE compares the input polytope against a new polytope that is derived from the input polytope by removing the input constraint. If the new polytope is contained within the input polytope then removing the constraint did not change the input polytope. In that case, the constraint is redundant and can be removed without changing the polytope. Note that this method of identifying redundant constraints is more powerful than testing whether the new constraint is implied by a single constraint. A constraint can be implied by a group of constraints but not by a single constraint. Our method allows to detect those cases as well.

Function ADDCONSTRAINTSIMP obtains as input a polytope $poly$ and a new constraint $newC$ to add to the polytope. The function first determines whether the new constraint to add is redundant. If this is the case, then the input polytope is not changed. If the new constraint is not redundant then it is inserted. In addition, we verify whether some of the old constraints become redundant due to the new constraint. For that purpose, we iterate over the old constraints and remove redundant ones.

We can construct RRs step by step by adding cutouts. Function SUBTRACTPOLYSIMP adds one new cutout and simplifies the region representation. We simplify RRs by discarding redundant cutouts. A cutout is redundant if it is covered by another cutout of the same region. Function SUBTRACTPOLYSIMP obtains as input a RR $rr$ and a new cutout $newCut$. It first verifies whether the new cut is covered by one of the old cutouts. While we compare a new constraint against all old constraints together in case of convex polytopes, we only compare pairs of cutouts. The reason is that the union of cutouts is not necessarily a convex polytope. Therefore we must restrict ourselves to pairwise comparisons between cutouts. If the new cutout is not covered by one of the old cutouts then the new cutout is added. In that case, we verify whether some of the old cutouts are covered by the new cutout. All covered cutouts are removed to simplify the region representation.

### 4.6.4 Complexity Analysis

The complexity of PQ, MQ, and MPQ algorithms depends heavily on the number of plans that are stored per table set. Prior work analyzing the complexity of PQ and MQ algorithms often considers the number of plans as random variable and derives upper bounds on its expected value [60, 59]. We adopt the same approach for analyzing the complexity of PWL-RRPA. We focus on the case of linear cost functions; the analysis can easily be generalized to PWL cost functions for a given number of pieces. Let $n_X$ be the number of parameters. The linear cost function of a plan $p$ can be described by a set of real-valued weights $w^p_{m,i} \in \mathbb{R}$ for $m \in \mathbb{M}$ and $i \in \{0,\ldots,n_X\}$. The cost of $p$ according to metric $m$ is given by the expression $w^p_{m,0} + \sum_{1\leq i} w^p_{m,i}x_i$ where $x_i$ designates the value of the $i$-th parameter. We say that a plan $p_1$ dominates a plan $p_2$ *parameter value independently* (p.v.i.) if $w^{p_1}_{m,i} \leq w^{p_2}_{m,i}$ for each metric $m$ and for each $i \in \{0,\ldots,n_X\}$. If $p_1$ dominates $p_2$ p.v.i. then $p_1$ dominates $p_2$ (according to the definition in Section 4.2) for all possible (positive) parameter values. Given a concrete parameter space, a plan $p_1$ dominating another plan $p_2$ p.v.i. is a sufficient (but not a necessary) condition for $p_1$ dominating $p_2$ in the entire parameter space. We now derive an upper bound on the expected number of Pareto plans assuming that plan cost weights are chosen randomly; we assume that weights of different plans and different weights for the same plan are chosen independently. All those assumptions are common in the complexity analysis of PQ and MQ algorithms [60, 59]. By $n_M = |\mathbb{M}|$, we designate the number of cost metrics.

**Theorem 19.** *The expected number of Pareto plans per table set is upper-bounded by* $2^{((n_X+1)\cdot n_M)}$.

*Sketch.* The cost function of a plan is described by $(n_X+1)\cdot n_M$ cost weights. Hence, a cost function can be thought of as a point in $(n_X+1)\cdot n_M$-dimensional space. Ganguly et al. [60] derive an upper bound of $2^l$ on the size of the cover set when choosing an unspecified number of points in $l$-dimensional space (see Theorem 3 in their publication). Setting $l = (n_X+1)\cdot n_M$, we can use that result to obtain an upper bound on the number of plans that are not dominated p.v.i. by any other plan. This is an upper bound on the number of plans that PWL-RRPA is expected to retain for any given table set after pruning (the bound is pessimistic since a plan that is not dominated p.v.i. may still be dominated in the entire concrete parameter space). □

The upper bound derived in Theorem 19 is consistent with prior results in the areas of PQ and MQ: the upper bound of $2^{n_M}$ on the expected number of plans derived for the case of $n_M$ cost metrics and no parameters (MQ) [60] corresponds to a specialization of our result. Our bound grows exponentially in the number of parameters which is in line with prior results on PQ [74] (tighter bounds require additional assumptions [59]). We denote our bound on the number of plans per table set by $n_P$ in the following, the number of scan and join operators by $n_O = |\mathbb{O}|$, and the number of tables by $n_Q = |Q|$. The function $\mathbf{lp}(a, b)$ represents the time for solving a linear program with matrix dimensions $a \times b$. An upper bound on the number of plans that PWL-RRPA generates per table set is given by $n_G = 2^{n_Q} n_P^2 n_O$.

**Lemma 11.** *Function* IsEmpty *has time complexity*
$O(n_M^{n_G} \mathbf{lp}(n_M n_G, n_X))$.

*Proof.* A cutout is a region in which one plan dominates another; a cutout is therefore defined by $n_M$ linear constraints. Comparing one plan to another one during pruning adds at most one cutout to its RR. The total number of cutouts per RR is therefore bounded by $n_G$. The time complexity of IsEmpty is dominated by the time for checking whether the union of polytopes is convex; Bemporad et al. [22] provide complexity results for their algorithm, we use them with $n_G$ as bound on the number of polytopes and $n_M$ as bound on the number of constraints per polytope. □

We denote the time complexity of IsEmpty by $T_{emp}$.

**Theorem 20.** PWL-RRPA *has time complexity*
$O(3^{n_Q} n_P^3 n_O T_{emp})$.

*Proof.* The time for emptiness checks dominates. Each newly generated plan is compared against $O(n_P)$ alternative plans which requires $O(n_P)$ emptiness checks. PWL-RRPA iterates over all subsets of $Q$. For a subset $q \subseteq Q$ containing $i = |q|$ tables, PWL-RRPA generates $O(2^i n_P^2 n_O)$ plans. Using $\sum_{i=1}^{n_Q} \binom{n_Q}{i} 2^i = 3^{n_Q}$ yields the total complexity. □

## 4.7 Experimental Evaluation

We experimentally evaluate PWL-RRPA in multiple scenarios. We first describe the experimental setup, then present the results, and finally discuss them.

**Experimental Setup.** We consider three scenarios. The first one is a Cloud scenario in which two cost metrics, execution time and monetary fees, are relevant. A parallel hash join and a single-node hash join are available. The parallel hash join requires to shuffle the input data in the network. Parallelization therefore increases the total amount of work (which is proportional to monetary cost) while it can decrease execution time in comparison to a single-node join if the input relations are sufficiently large. This shows that a tradeoff exists between execution time and monetary fees and a query plan that minimizes one does not necessarily minimize the other. Base tables are associated with equality predicates whose selectivites are represented by parameters; one parameter is required for each table with a predicate. Indices are available for each column with a predicate. This makes an index seek preferable for low selectivity while a complete table scan is better for non-selective predicates; as predicate selectivity is a parameter, plans must often be kept for both cases which makes the benchmark even more challenging.

Our second scenario focuses on approximate query processing. This time we consider the two cost metrics execution time and result precision. We assume that we can reduce execution time by sampling the input tables instead of processing them entirely. Sampling has however a negative impact on result precision. More precisely, we assume that we have for each base table the choice between taking a large and a small sample. Choosing a small sample reduces the amount of data that needs to be processed and therefore the execution time. We use a simple precision model where precision is proportional to the fraction of tuples of the true result (i.e., the result obtained without sampling) that we generate. This model is described in more detail in Chapter 2. Parameters represent again the selectivity of predicates on base tables. We consider two join operators: a hash join and a block-nested loop join. We do not consider indices in the second scenario.

Our third scenario examines tradeoffs between the execution time of a plan and its buffer space consumption. Execution time can often be reduced by dedicating additional buffer space to the execution of a plan. In cases where multiple queries execute concurrently and share a limited amount of memory, it is however crucial to find good tradeoffs between execution time and memory consumption for each single plan. We assume that two join operators are available, one of them consumes more buffer space but requires less time for sufficiently large join operands. Parameters represent again the selectivity of predicates and we do not consider indices.

We evaluate the performance of PWL-RRPA on randomly generated queries, using the generation method proposed by Steinbrunn [130] (and used recently in other publications [31]) to choose table cardinalities and join predicates; we assume that unique values occupy up to 10% of a table column. We separately evaluate the performance for star queries and for chain queries as the structure of the join graph is known to have significant impact on optimizer performance [130]. PWL-RRPA considers the full search space of bushy query plans but postpones Cartesian product joins as much as possible; this heuristic is commonly applied in state-of-the-art optimizers such as the Postgres optimizer[2]. Standard formulas are used to estimate join time, result precision, and buffer space consumption; monetary cost are calculated according to the pricing system of Amazon EC2[3] and the properties of the simulated cluster nodes such as main memory size correspond to the ones of the general purpose medium instance in EC2. PWL-RRPA was implemented in Java 1.7, using Gurobi 5.6[4] as linear program solver. All experiments were executed on a commodity iMac equipped with an i5-3470S processor with 2.9 GhZ and 16 GB of RAM.

**Experimental Results.** The goal of the following experiments is to show how optimization

[2]http://www.postgresql.org/
[3]http://aws.amazon.com/de/ec2/
[4]http://www.gurobi.com/

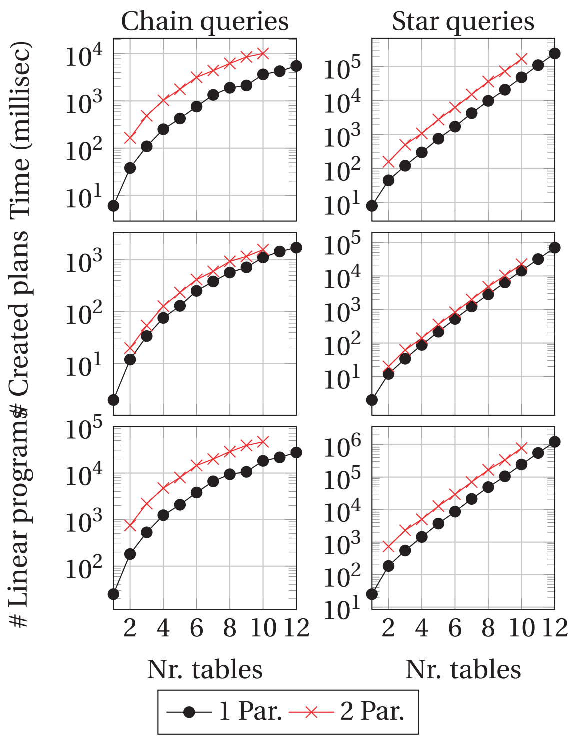


Figure 4.12 – Cloud scenario: optimization time, number of generated plans, and number of solved linear programs

time depends on query characteristics such as the number of tables, the number of parameters, and the join graph structure. We present our results for the Cloud scenario first. We experimented with up to 12 tables for one parameter and up to 10 tables for two parameters. Figure 4.12 shows optimization time, the number of generated plans (including partial plans and plans that were pruned during optimization), and the number of solved linear programs (LPs). Each data point corresponds to the median of 25 randomly generated test cases. All three metrics are clearly correlated and increase in the number of tables as well as in the number of parameters. The number of solved LPs is much higher than the number of generated plans since operations such as comparing plans during pruning or checking emptiness of a plan's RR all require to solve several LPs. As in traditional query optimization, optimizing chain queries is faster than optimizing star queries when avoiding Cartesian product joins [106].

Figure 4.13 shows our results for the second scenario (approximate processing). Here we experiment only with up to ten tables in case of one parameter and with up to eight tables in case of two parameters. Each data point represents the median of ten randomly generated test cases. We used less test cases than in the last scenario in order to reduce computational burden. The optimization times are generally higher than in the last scenario. This is explained by the fact that more plans are generated (which means that more linear programs need to be solved). We consider a search space of comparable size in both scenarios (we consider the same join orders and the same number of scan and join operators). The increase in the

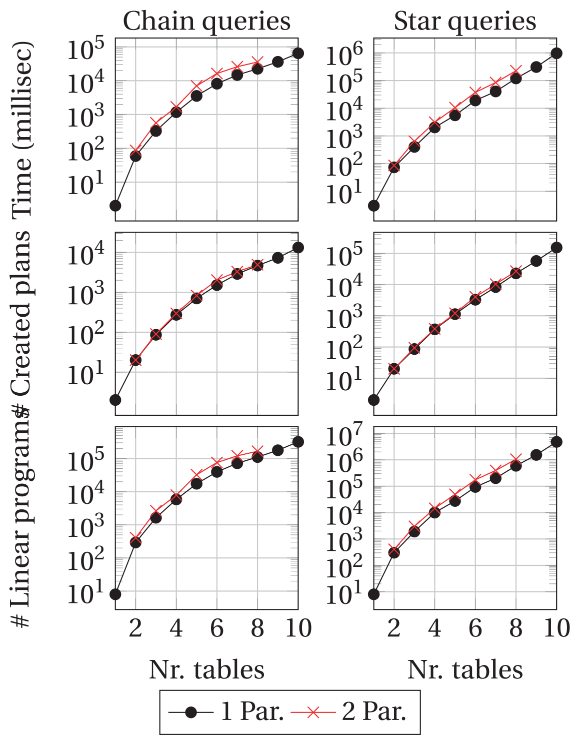


Figure 4.13 – Approximate processing scenario: optimization time, number of generated plans, and number of solved linear programs

number of generated plans is therefore due to the change of cost metrics. Execution time and result precision are strongly anti-correlated cost metrics: decreasing the sample size has always a positive impact on execution time but a negative impact on result precision (in the Cloud scenario, choosing the parallel join over the single-node join decreases execution time only if the input set is sufficiently large). Having strongly anti-correlated cost metrics generally tends to increase the number of optimal cost tradeoffs [120]. In our case, this means that the number of plans realizing optimal cost tradeoffs increases and so does the number of generated plans.

Figure 4.14 shows the experimental results for our third scenario. The general tendencies are similar to the previous scenarios: optimization times are higher for star queries and grow in the number of tables and parameters. Comparing optimization times between the three scenarios, we find that optimization times in the bufferspace scenario are situated in between the corresponding values of the previous scenarios. This can again be explained by the cost metrics. We first compare to the Cloud computing scenario. Parallelizing a join is only helpful if the operands are relatively large. In contrast to that, dedicating additional buffer space can speed up joins even if the input operands are of medium size (while it does not help for operands that are small enough to fit entirely into the originally dedicated join buffer). This explains that we obtain more optimal cost tradeoffs in the bufferspace scenario. On the other side, sampling can reduce execution time even for relatively small operands. For that reason,

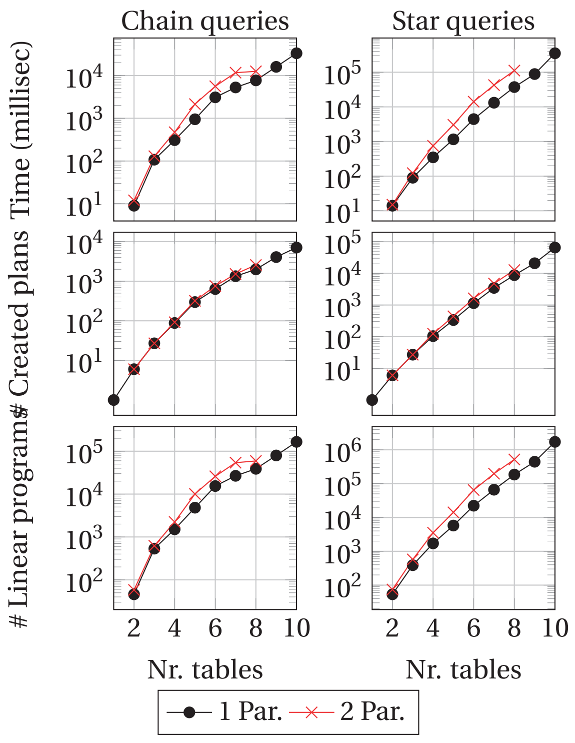


Figure 4.14 – Bufferspace-time tradeoff scenario: optimization time, number of generated plans, and number of solved linear programs

Table 4.2 – Average number of polytopes per cost function in different scenarios

| | **Chain** | | **Star** | |
|---|---|---|---|---|
| **Scenario** | 1 Par. | 2 Par. | 1 Par. | 2 Par. |
| **Cloud** | 2.6 | 3.3 | 2.7 | 3.2 |
| **Approximation** | 4.0 | 4.6 | 4.3 | 5.0 |
| **Bufferspace** | 3.2 | 3.3 | 3.1 | 3.9 |

the approximate processing scenario is the most difficult one.

Another noticeable difference between the three scenarios is the extent up to which adding a second parameter increases optimization time. While optimization time increases in all scenarios, the first scenario seems to be most sensitive to the addition of a second parameter. This is explained by the fact that we consider indices only in the first scenario. Adding parameters generally increases the number of optimal plans since different join orders can be optimal for different parameter values. But in the first scenario, adding parameters also increases the number of optimal scan operator selections: while an index scan is preferable for low selectivity values, a full scan is preferable for predicates with high selectivity.

We finally compare different scenarios in terms of the complexity of their cost functions. We measure that complexity as the average number of polytopes that is used to represent a cost function in one cost metric. Table 4.2 shows the results. The complexity of the cost functions correlates with optimization time. The approximate processing scenario has generally the most complicated cost functions. The Cloud scenario tends to have the simplest cost functions while their complexity increases the most by adding a second parameter.

**Discussion.** MPQ is a generalization of MQ and PQ and computationally expensive. MPQ happens however before run time and it pays off as it avoids run time query optimization altogether. Optimization times depend on the considered cost metrics and increase for anti-correlated cost metrics. Optimization time increases in the number of considered parameters. The growth depends on the scenario again. In the common case that parameters describe selectivity values, having operator selections that are particularly sensitive to input sizes leads to a more significant growth.

## 4.8 Conclusion

We introduced MPQ, a novel variant of query optimization that allows to consider multiple cost metrics and parameters. We presented a first algorithm for this problem and evaluated it in multiple scenarios. Our algorithm is exhaustive and guarantees to generate all relevant query plans. MPQ is a computationally expensive optimization problem. We plan to develop approximation algorithms for this problem in future work.

```
// Find a Pareto plan set for query $Q$
function GenericMPQ($Q$)
    // Initialize plan sets for base tables
    for $\langle q, p\rangle : q \in Q, p$ is plan for $q$ do
        Prune($\mathscr{P}, q, p$)
    end for
    // Consider table sets of increasing cardinality
    for $k \in 2..|Q|$ do
        // Iterate over table sets with given cardinality
        for $q \subseteq Q : |q| = k$ do
            $\mathscr{P}^q \leftarrow$GenerateParetoPlanSet($q$)
        end for
    end for
    return $\mathscr{P}^Q$
end function
// Generate Pareto plan set for joining $q$
function GenerateParetoPlanSet($q$)
    $\mathscr{P} \leftarrow \emptyset$
    // For all possible splits of table set $q$
    for $q_1, q_2 \subset q : q_1 \dot{\cup} q_2 = q$ do
        // For all sub-plans and operators
        for $p_1 \in \mathscr{P}^{q_1}, p_2 \in \mathscr{P}^{q_2}, o \in \mathbb{O}$ do
            // Construct new plan out of sub-plans
            $p_N \leftarrow$Combine($p_1, p_2, o$)
            // Accumulate cost of sub-plans
            $\mathbf{c}(p_N)$ =AccumulateCost($o, p_1, p_2$)
            // Prune with new plan
            Prune($\mathscr{P}, q, p_N$)
        end for
    end for
    return $\mathscr{P}$
end function
// Prune plan set $\mathscr{P}$ for query $q$ with new plan $p_N$
procedure Prune($\mathscr{P}, q, p_N$)
    // Check whether the new plan is relevant
    $\mathscr{R}^q(p_N) \leftarrow \mathbb{X}$
    for $p \in \mathscr{P}^q$ do
        // Update relevance region of new plan
        $\mathscr{R}^q(p_N) \leftarrow \mathscr{R}^q(p_N) \setminus$Dom($p, p_N$)
        // Check if relevance region became empty
        if $\mathscr{R}^q(p_N) = \emptyset$ then
            return // Do not insert new plan
        end if
    end for
    // If we arrive here, the new plan will be inserted
    // Discard irrelevant old plans
    for $p \in \mathscr{P}^q$ do
        // Update relevance region of old plan
        $\mathscr{R}^q(p) \leftarrow \mathscr{R}^q(p) \setminus$Dom($p_N, p$)
        // Check if relevance region became empty
        if $\mathscr{R}^q(p) = \emptyset$ then
            $\mathscr{P}^q \leftarrow \mathscr{P}^q \setminus \{p\}$ // Discard old plan
        end if
    end for
    // Insert new plan into Pareto plan set
    $\mathscr{P}^q \leftarrow \mathscr{P}^q \cup \{p_N\}$
end procedure
```

Algorithm 7 – The relevance region pruning algorithm for generic multi-objective parametric query optimization

```
// Input: relevance region $rr$, convex polytopes $polys$
// Effect: region $rr$ is reduced by $polys$
procedure SubtractPolys($rr, polys$)
    // Add polytopes to cutouts
    $rr.cutouts \leftarrow rr.cutout \cup polys$
end procedure
// Input: relevance region $rr$
// Output: true iff $rr$ is empty
function IsEmpty($rr$)
    // Check whether union of cutouts is convex
    if $(\cup_{C \in rr.cutouts} C)$ is convex then
        // Calculate convex polytope covered by cutouts
        $CutPoly \leftarrow$ polytope $(\cup_{C \in rr.cutouts} C)$
        // Check if cutouts cover whole parameter space
        if $\mathbb{X} \subseteq CutPoly$ then
            // Relevance region is empty
            return true
        end if
    end if
    // Cutouts do not cover whole parameter space
    return false
end function
```

Algorithm 8 – Elementary operations on relevance regions

```
// Input: a join operator $o$ and two plans $p_1$ and $p_2$
// Output: accumulated cost of executing $p_1$ and $p_2$
//     and joining their results using $o$
function AccumulateCost($o, p_1, p_2$)
    // Create new cost function
    $acCost \leftarrow$ new multi-obj. PWL cost func.
    // Iterate over all cost metrics
    for $m \in \mathbb{M}$ do
        // Initialize pieces of new cost function
        $newPcs \leftarrow \emptyset$
        // Iterate over cost function pieces of sub-plans
        for $fp_1 \in \mathbf{c}(p_1).comps[m].pieces$ do
            for $fp_2 \in \mathbf{c}(p_2).comps[m].pieces$ do
                // Intersect regions of the two pieces
                $r \leftarrow fp_1.reg \cap fp_2.reg$
                // Check if intersection is empty
                if $r \neq \emptyset$ then
                    // Add weight vectors
                    $\mathbf{w} \leftarrow fp_1.\mathbf{w} + fp_2.\mathbf{w} + o.\mathbf{w}$
                    // Add base costs
                    $b \leftarrow fp_1.b + fp_2.b + o.b$
                    // Construct new piece
                    $newPc \leftarrow$ new linear cost func. with
                   base cost $b$, weight $\mathbf{w}$, and region $r$
                    // Add new piece
                    $newPcs \leftarrow newPcs \cup \{newPc\}$
                end if
            end for
        end for
        $acCost.comps[m].pieces \leftarrow newPcs$
    end for
    return $acCost$
end function
// Input: two plans $p_1$ and $p_2$
// Output: a set of convex polytopes in the
//     parameter space where $p_1$ dominates $p_2$
function Dom($p_1, p_2$)
    // Calculate $p_1$'s dominant region for each metric
    for $m \in \mathbb{M}$ do
        // Initialize set of polytopes
        $polys_m \leftarrow \emptyset$
        // For all pairs of cost function pieces
        for $fp_1 \in \mathbf{c}(p_1).comps[m].pieces$ do
            for $fp_2 \in \mathbf{c}(p_2).comps[m].pieces$ do
                // Calculate intersection of regions
                $r \leftarrow fp_1.reg \cap fp_2.reg$
                // Calculate part where $p_1$ dominates $p_2$
                $rDom \leftarrow$ solutions to linear equations
            $(fp_1.\mathbf{w} - fp_2.\mathbf{w})^T \mathbf{x} \leq fp_2.b - fp_1.b, \mathbf{x} \in r$
                // Add polytope if not empty
                if $rDom \neq \emptyset$ then
                    $polys_m \leftarrow polys_m \cup \{rDom\}$
                end if
            end for
        end for
    end for
    // Combine results from different metrics
    return $\{\cap_{m \in \mathbb{M}} p_m | p_m \in polys_m\}$
end function
```

Algorithm 9 – Elementary operations on cost functions

```
// Input: polytope $poly$, linear constraint $C$
// Output: **true** iff removing $C$ does not change $poly$
function CanRemove($poly, C$)
    $newPoly \leftarrow$ copy of $poly$
    $newPoly.constr \leftarrow newPoly.constr \setminus \{C\}$
    return $newPoly \subseteq poly$
end function
// Input: polytope $poly$, linear constraint $newC$
// Effect: add constraint and simplify polytope
procedure AddConstraintSimp($poly, newC$)
    // Verify whether new constraint is redundant
    $poly.constr \leftarrow poly.constr \setminus \{newC\}$
    if CanRemove($poly, newC$) then
        // New constraint is redundant: remove it
        $poly.constr \leftarrow poly.constr \setminus \{newC\}$
        return
    end if
    // Remove old constraints that are implied by $newC$
    for all $oldC \in poly.constr \setminus \{newC\}$ do
        if CanRemove($poly, oldC$) then
            $poly \leftarrow poly \setminus \{oldC\}$
        end if
    end for
end procedure
// Input: relevance region $rr$, convex polytope $newCut$
// Effect: reduce $rr$ by $newCut$ and simplify $rr$
procedure SubtractPolySimp($rr, newCut$)
    // Verify whether new cutout is redundant
    for all $oldCut \in rr.cutouts$ do
        if $newCut \subseteq oldCut$ then
            // No need to add the new cutout
            return
        end if
    end for
    // The new cut is not redundant and will be added
    // Remove old cutouts that are covered by $newCut$
    for all $oldCut \in rr.cutouts$ do
        if $oldCut \subseteq newCut$ then
            $rr.cutouts \leftarrow rr.cutouts \setminus \{oldCut\}$
        end if
    end for
    // Add new cutout
    $rr.cutouts \leftarrow rr.cutouts \cup \{newCut\}$
end procedure
```

Algorithm 10 – Methods for simplifying the representations of parameter space regions

# 5 Randomization

The approaches presented in the last chapters work well for medium-sized queries. We need however different algorithms in order to deal with large queries. In this chapter, we will see a randomized algorithm that is able to handle queries with hundreds of joins. This algorithm is the first randomized algorithm for multi-objective query optimization. It combines algorithmic ideas that are typically used in different research branches in query optimization. The algorithm is tailored to the multi-objective query optimization problem. We will see that it outperforms randomized general-purpose algorithms for multi-objective optimization as well as traditional algorithms for query optimization significantly. We will also see that the randomized algorithm handles query sizes that cannot be treated anymore using the approximation schemes from the previous chapters.

## 5.1 Introduction

So far, exhaustive algorithms [60, 139] and several approximation schemes (described in the previous chapters) have been proposed to solve the generic multi-objective query optimization problem. The exhaustive algorithms formally guarantee to find the full Pareto frontier while the approximation schemes formally guarantee to approximate the Pareto frontier with a certain minimum precision. Those quality guarantees come at a cost in terms of optimizer performance: all existing algorithms for multi-objective query optimization have at least exponential time complexity in the number of tables (potentially higher depending on the number of Pareto plans). This means that they cannot be applied for queries with elevated number of tables.

For the traditional query optimization problem with one cost metric, there is a rich body of work proposing heuristics and randomized algorithms [135, 131, 76, 23]. Those algorithms offer no formal quality guarantees on how far the generated plans are from the theoretical optimum but often generate good plans in practice. They have polynomial complexity in the number of tables and can be applied to much larger queries than exhaustive approaches. Up to date, corresponding approaches for multi-objective query optimization are missing entirely (and we will show later that algorithms for traditional query optimization perform poorly for the multi-objective case). In this chapter we close that gap and present the first randomized algorithm for multi-objective query optimization with polynomial time complexity.

Existing algorithms for single- or multi-objective query optimization typically exploit only one out of two fundamental insights about the query optimization problem: Dynamic programming based algorithms [116, 91, 137, 138, 139] exploit its decomposability, i.e. the fact that a query optimization problem can be split into smaller sub-problems such that optimal solutions (query plans) for the sub-problems can be combined into optimal solutions for the original problem. Randomized algorithms such as iterative improvement, simulated annealing, or two-phase optimization exploit a certain near-convexity (also called well shape [75]) of the standard cost functions when using suitable neighboring relationships in the query plan space. There is no reason why both insights shouldn't be exploited within the same algorithm and we do so: our algorithm improves plans using a multi-objective generalization of hill climbing, thereby exploiting near-convexity. It also maintains a plan cache storing partial Pareto-optimal plans generating potentially useful intermediate results. Newly generated plans are decomposed and dominated sub-plans are replaced by partial plans from the cache. Therefore we exploit decomposability as well.

Our algorithm is iterative and performs the following steps in each iteration. First, a query plan is randomly generated. Second, the plan is improved using local search until a local optimum is reached. Third, based on the locally optimal plan we restrict the plan space to plans that are similar in certain aspects (we provide details in the following paragraphs). We approximate the Pareto plan set within that restricted plan space. For that approximation, we might re-use partial plans that were generated in prior iterations if they realize a better cost tradeoff than the corresponding sub-plans of the locally optimal plan.

For the second step, we use a multi-objective version of hill climbing that exploits several properties of the query optimization problem to reduce time complexity significantly compared to a naive version. First, we exploit the multi-objective principle of optimality for query optimization [60] stating that replacing a sub-plan by another sub-plan whose cost is not Pareto-optimal cannot improve the cost of the entire plan. This allows to quickly discard local mutations that will not improve the overall plan. Second, we exploit that query plans can be recursively decomposed into sub-plans for which local search can be applied independently. This allows to apply many beneficial mutations simultaneously in different parts of the query tree and therefore reduces the number of complete query plans that need to be generated on the path from the random plan to a local optimum. Those optimizations reduce the time complexity comparing with a naive version and we found them to be critical to achieve good optimizer performance.

For the third step, we restrict the plan space to plans that generate a similar set of intermediate results as the locally optimal plan that results from the second step. We consider plans that use the same join order as the locally optimal plan but different operator combinations. Also, we consider the possibility to replace sub-plans by plans from a cache (those plans can use a different join order than the locally optimal plan). The cache stores non-dominated partial plans for each potentially useful intermediate result we encountered during optimization so far. Finding the full Pareto plan set even within the restricted plan space may lead to prohibitive computational cost (the number of Pareto plans within the restricted space may grow exponentially in the number of query tables). We therefore approximate the Pareto plan set by a subset of plans (whose size grows polynomially in the number of query tables) realizing representative cost tradeoffs. The precision of that approximation is slowly refined over the iterations. This enables our algorithm to quickly find a coarse-grained approximation of the full Pareto plan set for the given query; at the same time, as we refine precision, the approximation converges to the real Pareto set as iterations continue.

The insights underlying the design of our algorithm are the following: on the one hand, we observe that the same join order can often realize many Pareto-optimal cost tradeoffs when using different operator configurations. This is why we approximate in the third step a Pareto frontier based on a restricted set of join orders. On the other hand, the full Pareto frontier cannot be covered using only one join order. This is why we generate new plans and join orders in each iteration.

We analyze our randomized algorithm experimentally, using different cost metrics, query graph structures, and query sizes. We compare against dynamic programming based approximation schemes that were previously proposed for multi-objective query optimization. While approximation schemes are preferable for small queries, we show that only randomized algorithms can handle larger query sizes. We evaluate our algorithm with queries joining up to 100 tables considering an unconstrained bushy plan space. Even in case of one cost metric, dynamic programming based approaches do not scale to such search space sizes. We also compare our algorithm against other randomized algorithms: the non-dominated sort genetic algorithm 2 (NSGA-II) [49] is a very popular multi-objective optimization algorithm for the number of plan cost metrics that we consider in our experiments. Genetic algorithms have been very successful for traditional query optimization [23] so a comparison against NSGA-II (using the combination and mutation operators proposed for traditional query optimization) seems interesting. We also compare our algorithm against other multi-objective generalizations of well known randomized algorithms for traditional query optimization such as iterative improvement, simulated annealing, and two-phase optimization [131]. Our randomized algorithm outperforms all competitors significantly, showing that the combination of local search with plan decomposition is powerful.

We analyze the time complexity of our algorithm and show that each iteration has expected polynomial complexity. Our analysis includes in particular a study of the expected path length from a random plan to the nearest local optimum. Based on a simple statistical model of plan cost distributions, we can show that the expected path length grows at most linearly in the number of query tables.

In summary, the original scientific contributions of this chapter are the following:

- We present the first polynomial time algorithm for multi-objective query optimization. A randomized algorithm that exploits several properties of the query optimization problem.
- We analyze that algorithm formally, showing that each iteration (resulting in at least one query plan) has polynomial complexity in the number of query tables.
- We evaluate our algorithm experimentally against previously published approximation schemes for multi-objective query optimization and several randomized algorithms. We show that our algorithm outperforms the other algorithms over a wide range of scenarios.

The remainder of this chapter is organized as follows. We give an overview of related work in Section 5.2. In Section 5.3, we introduce the formal model used in pseudo-code and formal analysis. We introduce the first randomized algorithm for multi-objective query optimization in Section 5.4 and analyze its complexity in Section 5.5. In Section 5.6, we experimentally evaluate our algorithm in comparison with several baselines. We see additional experimental results in Section 5.7, showing that the results from Section 5.6 generalize over many scenarios.

## 5.2 Related Work

Most work in query optimization treats the single-objective case, meaning that query plans are compared according to only one cost metric (usually execution time) [23, 31, 116, 131, 135, 141]. This problem model is however often insufficient: in a cloud scenario, users might be able to reduce query execution time when willing to pay more money for renting additional resources from the cloud provider [88]. On systems that process multiple queries concurrently, the tradeoff between the amount of dedicated system resources and query execution time needs to be considered [137]. In those and other scenarios, query optimization becomes a multi-objective optimization problem.

Query optimization algorithms that have been designed for the case of one cost metric cannot be applied to the multi-objective case. They return only one optimal plan while the goal in multi-objective query optimization is usually to find a set of Pareto-optimal plans [107, 139, 138]. This allows in particular to let users choose their preferred cost tradeoff out of a visualization of the available tradeoffs (see Chapter 3). We will use several variations of single-objective randomized query optimization algorithms as baselines for our experiments. Note that mapping multi-objective optimization into a single-objective optimization problem using a weighted sum over different cost metrics with varying weights will not yield the Pareto frontier but at most a subset of it (the convex hull).

We have described multiple algorithms for multi-objective query optimization in the previous chapters. They are based on dynamic programming and have all exponential complexity in the number of query tables. This means that they do not scale to large query sizes. In traditional single-objective optimization, randomized algorithms and heuristics are used for query sizes that cannot be handled by exhaustive approaches. No equivalent is currently available for multi-objective query optimization. In this work we close that gap and propose the first polynomial time heuristic for multi-objective query optimization.

One of the most popular randomized algorithms for single-objective query optimization is the genetic algorithm [23]. Also, multi-objective genetic algorithm variants are very popular for multi-objective optimization in general [42]. It seems therefore natural to use the crossover and mutation operators that have been proposed for traditional query optimization within a multi-objective genetic algorithm variant. We implemented a version of the widely used non-dominated sort genetic algorithm II (NSGA-II) [49] for our experiments.

We focus on query optimization in the traditional sense, i.e. the search space is the set of available join orders and the selection of scan and join operators. This distinguishes our work for instance from work on multi-objective optimization of workflows [88, 126] which does not consider alternative join orders. Prior work by Papadimitriou and Yannakakis [107] also aims at optimizing operator selections for a fixed join order and addresses therefore a different problem than us. We focus on generic multi-objective query optimization as in the previous chapters and our algorithm is not bound to specific combinations of cost metrics or scenarios such as precision-time [12] or energy-time tradeoffs [147].

## 5.3 Formal Model

Our notation is similar to the ones used in previous chapters. Nevertheless, we introduce notation from scratch to make the current chapter self-contained.

A query $q$ is modeled as a set of tables that need to be joined. This query model is simplistic but often used in query optimization [135, 60, 137]; extending a query optimization algorithm that optimizes queries represented as table sets to more complex query models is standard [116]. A query plan $p$ for a query $q$ specifies how the data described by the query can be generated by a series of scan and join operations. The plan describes the join order and the operator implementation used for each scan and join. We write $\textsc{ScanPlan}(q, op)$ to denote a plan scanning the single table $q$ ($|q| = 1$) using scan operator $op$. We write $\textsc{JoinPlan}(outer, inner, op)$ to denote a join plan that joins the results produced by an outer plan $outer$ with the results produced by an inner plan $inner$ using join operator $op$.

We denote by $p.rel$ the set of tables joined by a plan $p$. It is $\textsc{ScanPlan}(q, op).rel = q$ and $\textsc{JoinPlan}(po, pi, op).rel = po.rel \cup pi.rel$. For join plans we denote by $p.outer$ the outer plan and by $p.inner$ the inner plan. The property $p.isJoin$ yields **true** for join plans (where $|p.rel| > 1$) and **false** for scan plans (where $|p.rel| = 1$).

We compare query plans according to their execution cost. We consider multiple cost metrics that might for instance include monetary fees (in a cloud scenario [88]), energy consumption [147], or various metrics of system resource consumption such as the number of used cores or the amount of consumed buffer space [137] in addition to execution time. We consider cost metrics in the following meaning that a lower value is preferable. It is straight forward to transform a quality metric on query plans such as result precision [12] into a cost metric (e.g., result precision can be transformed into the precision loss cost metric as shown in Chapter 2). Hence we study only cost metrics in the following without restriction of generality.

We denote by $p.cost \in \mathbb{R}^l$ the cost vector associated with a query plan. Each vector component represents cost according to a different cost metric out of $l$ cost metrics. We focus on optimization and assume that cost models for all considered cost metrics are available. The algorithms that we discuss in this chapter are generic and can be applied to a broad set of plan cost metrics.

In the special case of one cost metric, we say that plan $p_1$ is better than plan $p_2$ if the cost of $p_1$ is lower. Pareto-dominance is the generalization to multiple cost metrics. In case of multiple cost metrics, we say that plan $p_1$ dominates $p_2$, written $p_1 \preceq p_2$ if $p_1$ has lower or equivalent cost to $p_2$ according to each considered cost metric. We say that $p_1$ strictly dominates $p_2$, written $p_1 \prec p_2$, if $p_1 \preceq p_2$ and $p_1.cost \neq p_2.cost$, meaning that $p_1$ has lower or equivalent cost than $p_2$ according to each metric and lower cost in at least one metric. We apply the same terms and notations to cost vectors in general, e.g. we write $c_1 \preceq c_2$ for two cost vectors $c_1$ and $c_2$ to express that there is no cost metric for which $c_1$ contains a higher cost value than $c_2$.

Considering a set $P$ of alternative plans generating the same results, we call each plan $p \in P$ Pareto-optimal if there is no other plan $\widetilde{p} \in P$ that strictly dominates $p$. For a given query, the Pareto plan set is the set of plans for $q$ that are Pareto-optimal within the set of all possible query plans for $q$. The full Pareto plan set is often too large to be calculated in practice. This is why we rather aim at approximating the Pareto plan set. The following definitions are necessary to establish a measure of how well a given plan set approximates the real Pareto set.

A plan $p_1$ approximately dominates a plan $p_2$ with approximation factor $\alpha \geq 1$, written $p_1 \preceq_\alpha p_2$, if $p_1.cost \preceq \alpha \cdot p_2.cost$. This means that the cost of $p_1$ is not higher than the cost of $p_2$ by more than factor $\alpha$ according to each cost metric. Considering a set $P$ of plans, an $\alpha$-approximate Pareto plan set $P_\alpha \subseteq P$ is a subset of $P$ such that $\forall p \in P \exists \widetilde{p} : \widetilde{p} \preceq p$, i.e. for each plan $p$ in the full set there is a plan in the subset that approximately dominates $p$. The Pareto frontier (or approximate Pareto frontier) are the cost vectors of the plans in the Pareto set (or approximate Pareto set).

The goal of multi-objective query optimization is to find $\alpha$-approximate Pareto plan sets for a given input query $q$. We compare different incremental optimization algorithms in terms of the $\alpha$ values (i.e., how well the plan set generated by those algorithms approximates the real Pareto set) that they produce after certain amounts of optimization time.

```
1:  // Returns approximate Pareto plan set for query $q$
2:  function RandomMOQO($q$)
3:      // Initialize partial plan cache and iteration counter
4:      $P \leftarrow \emptyset$
5:      $i \leftarrow 1$
6:      // Refine frontier approximation until timeout
7:      while No Timeout do
8:          // Generate random bushy query plan
9:          $plan \leftarrow$RandomPlan($q$)
10:         // Improve plan via fast local search
11:         $optPlan \leftarrow$ParetoClimb($plan$)
12:         // Approximate Pareto frontier
13:         $P \leftarrow$ApproximateFrontiers($optPlan, P, i$)
14:         $i \leftarrow i + 1$
15:     end while
16:     return $P[q]$
17: end function
```

Algorithm 11 – Main function.

## 5.4 Algorithm Description

We describe a randomized algorithm for multi-objective query optimization. Section 5.4.1 describes the main function, the following two subsections describe sub-functions. Section 5.4.2 describes a multi-objective hill climbing variant that executes multiple plan transformations in one step for maximal efficiency. Section 5.4.3 describes how we generate a local Pareto frontier approximation for a given join order, using non-dominated partial plans from a plan cache and trying out different operator configurations.

### 5.4.1 Overview

Algorithm 11 is the main function of our optimization algorithm. The input is a query $q$ and the output a set of query plans that approximate the Pareto plan set for $q$.

Our algorithm is iterative and refines the approximation of the Pareto frontier in each iteration. Each iteration consists of three principal steps: a random query plan is generated (local variable $plan$ in the pseudo-code), it is improved via a multi-objective version of hill climbing, and afterwards the improved plan (local variable $optPlan$) is used as base to generate a local Pareto frontier approximation. For the latter step, a plan cache (local variable $P$ in the pseudo-code) is used that stores for each intermediate result (i.e., a subset $s \subseteq q$ of joined tables) that we encountered so far a set of non-dominated partial plans. For the locally optimal plan resulting from hill climbing, we consider plans that can be obtained by varying the operator configurations but not the join order. In addition, we consider replacing sub-plans by non-dominated partial plans from the plan cache.

Within that restricted plan space, we do not search for the entire Pareto frontier as its size can be exponential in the number of query tables. Instead, we search for an approximation that has guaranteed polynomial size in the number of query tables. The precision of those approximations is refined with increasing number of iterations: our goal is to obtain a coarse-grained approximation of the entire Pareto frontier quickly so we start with a coarse approximation precision to quickly explore a large number of join orders. In later iterations, the precision is refined to allow to better exploit the set of join orders that was discovered so far. As the frontier approximation precision depends on the iteration number, Function APPROXIMATEFRONTIERS obtains the iteration counter $i$ as input parameter in addition to the locally optimal plan and the plan cache. All non-dominated partial plans generated during the frontier approximation are inserted into the plan cache $P$ and might be reused in following iterations.

After the timeout, the result plan set is contained in the plan cache and associated with table set $q$, the entire query table set (we use the array notation $P[q]$ to denote the set of cached Pareto plans that join table set $q$). Note that we can easily use different termination conditions than a timeout. In particular, in case of interactive query optimization where users choose an execution plan based on a visualization of available cost tradeoffs (see Chapter 3), optimization ends once the user selects a query plan for execution from the set of plans generated so far.

Our algorithm exploits two ideas that have been very successful in traditional query optimization but are typically used in separate algorithms: our algorithm exploits a certain near-convexity of typical plan cost functions [75] by using local search (function PARETOCLIMB) to improve query plans. It exploits however at the same time that the query optimization problem can be decomposed into sub-problems, meaning that Pareto-optimal plans joining table subsets can be combined into Pareto-optimal plans joining larger table sets. It is based on the insight that the same join order often allows to construct multiple Pareto-optimal cost tradeoffs by varying operator implementations but takes into account at the same time that not all optimal cost tradeoffs can be found considering only one join order. We describe the two principal sub-functions of Algorithm 11, function PARETOCLIMB and function APPROXIMATEFRONTIERS, in more detail in the following subsections.

Note finally that the algorithm can easily be adapted to consider different join order spaces (e.g., left-deep plans) by exchanging the random plan generation method and the set of considered local transformations.

### 5.4.2 Pareto Climbing

Algorithm 12 shows the pseudo-code of function PARETOCLIMB (and of several auxiliary functions) that is used by Algorithm 11 to improve query plans via local search. The input is a query plan $p$ to improve and the output is a locally optimal query plan that was reached from the input plan using a series of local transformations.

Hill climbing was already used in traditional query optimization [135] but our hill climbing variants differ from the traditional versions in several aspects that are discussed in the following. A first obvious difference is that we consider multiple cost metrics on query plans while traditional query optimization considers only execution time. In case of one cost metric, hill climbing moves from one plan to a neighbor plan (i.e., a plan that can be reached via a local transformation [131]) if the neighbor plan has lower cost than the first one. Our multi-objective version moves from a first plan to a neighbor if the neighbor strictly dominates the first one in the Pareto sense, i.e. the second plan has lower or equivalent cost according to all metrics and lower cost according to at least one.

In principle there can be multiple neighbors that strictly dominate the start plan while different neighbors do not necessarily dominate each other. In such cases, it cannot be determined which neighbor is the best one to move to and all neighbors might be required for the full Pareto frontier. In order to avoid a combinatorial explosion, we still chose to arbitrarily select one neighbor that strictly dominates the start plan instead of opening different path branches during the climb. The goal of function PARETOCLIMB within Algorithm 11 is to find one good plan while function APPROXIMATEFRONTIERS (which is discussed in the next subsection) will take care of exploring alternative cost tradeoffs.

Unlike most prior hill climbing variants used for traditional query optimization [135, 131], we chose to exhaustively explore all neighbor plans in each step of the climb instead of randomly sampling a subset of neighbors. We initially experimented with random sampling of neighbor plans which led however to poor performance. We believe that this is due to the fact that dominating neighbors become more and more sparse as the number of considered cost metrics grows. Using the simple statistical model that we introduce in Section 5.5, the probability of finding a dominating neighbor decreases exponentially in the number of cost metrics. Furthermore, sampling introduces overhead and makes it harder if not impossible to use the techniques for complexity reduction that we describe next.

Reducing the time complexity of local search as much as possible is crucial as function PARETO-CLIMB is called in each iteration of Algorithm 11. We exploit properties of the multi-objective query optimization problem in order to make our implementation of local search much more efficient than a naive implementation. A naive hill climbing algorithm iterates the following steps until a local optimum is reached: in each step, it traverses all nodes of the current query plan tree and applies to each node a fixed set of local mutations. For each mutation and plan node, a new complete neighbor plan is created and its cost is calculated. Based on that cost, the next plan is selected among the neighbors. This naive approach has per step quadratic complexity in the number of plan nodes (which is linear in the number of query tables).

For a first improvement, we can exploit the principle of optimality for multi-objective query optimization [60]. After applying a local transformation to one specific node in the query tree, it is not always necessary to calculate the cost of the completed plan (at the tree root) in order to determine whether that mutation reduces the plan cost. Due to the principle of optimality, improving a sub-plan cannot worsen the entire plan (we assume for the current discussion that all alternative plans produce the same data representation, neglecting for instance the impact of interesting orders [116]). In many cases, reducing the cost of a sub-plan even guarantees that the cost of the overall plan decreases as well[1]. This means that we can at least exclude that a certain mutation reduces the cost of the entire plan if it worsens the cost of the sub-plan to which it was applied. As the cost of the sub-plan can be recalculated in constant time (we treat the number of cost metrics as a constant as in the previous chapters), this simple optimization already reduces the complexity per step from quadratic in the number of tables to linear whenever the chances that a local cost improvement does not yield a global improvement are negligible.

While the last optimization reduces the complexity per climbing step, the optimization discussed next tends to reduce the number of steps required to reach the nearest local optimum. Query plans are represented as trees and we can simultaneously apply mutations in independent sub-trees; it is not necessary to generate complete query trees after each single mutation which is what the naive hill climbing variant does. If we reduce the cost of several sub-trees simultaneously then the cost of the entire plan cannot worsen either due to the principle of optimality. Applying multiple beneficial transformations in different parts of the query tree simultaneously reduces the number of completed query trees that are created on the way to the local optimum.

Note that all discussed optimization are also useful to improve the efficiency of local search for traditional query optimization with one cost metric. Local search has already been used for traditional query optimization but we were not able to find any discussion of the aforementioned issues in the literature while they have significant impact on the performance (the second optimization alone reduced the average time for reaching local optima from randomly selected plans by over one order of magnitude for queries with 50 tables in a preliminary benchmark).

Algorithm 12 integrates all of the aforementioned optimizations. Function ParetoClimb performs plan transformations until the plan cannot be improved anymore, i.e. there is no neighbor plan with dominant cost. Function ParetoStep realizes one transformation step. It may return multiple Pareto-optimal plan mutations that produce data in different representations (e.g., materialized versus non-materialized). We must do so since sub-plans producing different data representations cannot be compared as the data representation can influence the cost (or applicability) of other operations higher-up in the plan tree. We assume that the standard mutations for bushy query plans [131] are considered for each node in the plan tree. Function ParetoStep might however mutate multiple nodes in the tree during one call: when treating join plans then the outer and the inner sub-plan are both replaced by dominant mutations via a recursive call. We try out each combination of potentially improved sub-plans and try all local transformations for each combination. The

[1]This is guaranteed for cost metrics such as energy consumption, monetary cost, precision loss, and execution time when considering plans without parallel branches [137] where the cost of a plan is calculated as weighted sum or product over the cost of its sub-plans. It might not hold in special cases (e.g., when replacing a sub-plan that is not on the critical path in a parallel execution scenario by a faster one).

resulting plans are pruned such that only one non-dominated plan is kept for each possible output data representation. Function BETTER is used during pruning to compare query plans and returns **true** if and only if a first plan produces the same output data representation as a second (tested using function SAMEOUTPUT) and the cost vector of the first plan strictly dominates the one of the second.

The following example illustrates how Algorithm 12 works.

**Example 12.** *We assume for simplicity that only one plan cost metric is considered, that we have only one scan and join operator implementation such that only join order matters, and that the only mutation is the exchange of outer and inner join operands. We invoke function* PARETOCLIMB *on an initial query plan* $(S \bowtie T) \bowtie (R \bowtie U)$. *Function* PARETOCLIMB *invokes function* PARETOSTEP *to improve the initial plan. The root of the initial query plan is a join between* $(S \bowtie T)$ *and* $(R \bowtie U)$. *Function* PARETOSTEP *tries to improve the two operands of that join via recursive calls before considering mutations at the plan root. Hence one instance of* PARETOSTEP *is invoked to improve the outer operand* $(S \bowtie T)$ *and another instance is invoked to improve the inner operand* $(R \bowtie U)$. *The instance of function* PARETOSTEP *treating* $(S \bowtie T)$ *spawns new instances for improving the two join operands* $S$ *and* $T$. *As we consider no scan operator mutations here, those invocations do not have any effect. After trying to improve* $S$ *and* $T$, *the instance treating* $(S \bowtie T)$ *tries the mutation* $(T \bowtie S)$. *Assume that this mutation reduces cost. Then the improved sub-plan* $(T \bowtie S)$ *will be returned to the instance of function* PARETOSTEP *treating the initial plan. Assume that the sub-plan* $(R \bowtie U)$ *cannot be improved by exchanging outer and inner join operand. Then the top-level instance of function* PARETOSTEP *will try the mutation* $(R \bowtie U) \bowtie (T \bowtie S)$ *and compare its execution cost to the cost of* $(T \bowtie S) \bowtie (R \bowtie U)$. *The cheaper plan is returned to function* PARETOCLIMB *which detects that the initial plan has been improved. This function performs another iteration of the main loop as changing the initial plan can in principle enable new transformations that yield further improvements. This is not possible in the restricted setting of our example and hence function* PARETOCLIMB *terminates after two iterations.*

### 5.4.3 Frontier Approximation

The goal of Pareto climbing is to find one plan that is at least locally Pareto-optimal. Having such a plan, it is often possible to obtain alternative optimal cost tradeoffs by varying the operator implementations while reusing the same join order[2]. We exploit that fact in function APPROXIMATEFRONTIERS whose pseudo-code is shown in Algorithm 13.

Function APPROXIMATEFRONTIERS obtains a query plan $p$ whose join order it exploits, the

[2]More precisely, this is possible when choosing an appropriate formalization of the plan space: when considering tradeoffs between buffer space consumption and execution time, we can for instance introduce different versions of the standard join operators that work with different amounts of buffer space. When considering tradeoffs between result precision and execution time in approximate query processing, we might introduce different scan operator versions associated with different sample densities. In a cloud scenario, we can introduce operator versions that are associated with different degrees of parallelism, allowing to trade monetary cost for execution time.

plan cache $P$ mapping intermediate results to non-dominated plans generating them, and the iteration counter $i$ (counting iterations of the main loop in Algorithm 11) as input. The output is an updated plan cache in which the non-dominated plans generated in the current invocation have been inserted for the corresponding intermediate results.

Function ApproximateFrontiers starts by choosing the approximation factor $\alpha$ which depends on the iteration number $i$. A higher approximation factor reduces the time required for approximation but a lower approximation factor yields a more fine-grained approximation. We will see in Section 5.5 that ApproximateFrontiers has dominant time complexity within each iteration so a careful choice of the approximation factor is crucial. Multi-objective query optimization can be an interactive process in which alternative cost tradeoffs are visualized to the user such that he can make his choice (also see Chapter 3). Especially in such scenarios it is beneficial to rather obtain a coarse-grained approximation of the entire Pareto frontier quickly than to obtain a very fine-grained approximation of only a small part of it. As approximating the entire Pareto frontier requires in general considering many join orders (see our remark at the end of this subsection), we do not want to spend too much time at the beginning exploiting one single join order. For that reason we start with a coarse-grained approximation factor that is reduced as iterations progress. We have tried different formulas for choosing $\alpha$ and the formula given in the pseudo-code worked best for a broad range of scenarios.

Having chosen the approximation precision, the function approximates the Pareto frontier for scan plans by trying out all available scan operators on the given table. For join plans, a frontier is first generated for the outer and inner operand using recursive calls (so we traverse the query plan tree in post-order). After that, we consider each possible pair of a plan from the outer frontier with a plan from the inner frontier and each applicable join operator and generate one new plan for each such combination. Newly generated plans are pruned again but the definition for the pruning function differs from the one we used in Algorithm 12. For the same output data properties, the pruning function in Algorithm 13 might now keep multiple plans that realize different optimal cost tradeoffs. However, in contrast to the previous pruning variant, new plans are only inserted if their cost cannot be approximated by any of the plans already in the pruned plan set. We will see in Section 5.5 that this pruning function guarantees that the number of plans stored for a single table set is bounded by a polynomial in the number of query tables.

Note that function ApproximateFrontiers does not only try different operator combinations for a given join order. Instead, we consider all non-dominated plans that are cached for the intermediate results generated by the input plan $p$. The plans we consider might have been inserted into the plan cache in prior iterations of the main loop and might therefore use different join orders. The plan cache is our mean of sharing information across different iterations of the main loop. As iterations continue, the content of the plan cache will more and more resemble the set of partial plans that is generated by the approximation schemes presented in Chapter 2 and Chapter 3. However, instead of approximating the frontier for each possible intermediate result (implying exponential complexity), we only ever treat table sets that are used by locally Pareto-optimal plans.

Note finally that it is in general not possible to obtain all Pareto optimal query plans by varying operators for a fixed join order. Considering again the tradeoff between buffer space and execution time, a left-deep plan using pipelining might for instance minimize execution time while a non-pipelined bushy plan achieves the lowest buffer footprint. Or in case of approximate query processing, different tradeoffs between result precision and execution time can be achieved by varying the sample size generated by scan operators so the output size for each table and hence the optimal join order depends on the operator configurations. This means that we need to consider different join orders in order to obtain the full Pareto set; we cannot decompose query optimization into join order selection followed by operator selection. Our algorithm respects that fact.

## 5.5 Complexity Analysis

We analyze the time and space complexity of the algorithm presented in Section 5.4. We call that algorithm short RMQ for randomized multi-objective query optimizer in the following.

We denote by $n$ the number of query tables to be joined. The number of cost metrics according to which query plans are compared is specified by $l$. We assume that $n$ is variable while $l$ is a constant as in the previous chapters. As in prior work [60], we simplify the following formulas by assuming only one join operator while the generalization is straight-forward. We also neglect interesting orders for the following analysis such that query plans joining the same tables are only compared according to their cost vectors.

We analyze the time complexity of one iteration. Each iteration consists of three steps (random plan generation, local search, and frontier approximation); we analyze the complexity of each of those steps in the following. Random plan sampling (function RANDOMPLAN in Algorithm 11) can be implemented with linear complexity as shown by the following lemma.

**Lemma 12.** *Function* RANDOMPLAN *executes in $O(n)$ time.*

*Proof.* Query plans are labeled binary trees whose leaf nodes represent input tables. Quiroz shows how to generate binary trees in $O(n)$ [112]. Labels representing input tables and operators can be selected in $O(n)$ as well. Estimating the cost of a query plan according to all cost metrics is in $O(ln) = O(n)$ time (as $l$ is a constant). □

Next we analyze the complexity of local search, realized by function PARETOCLIMB in Algorithm 11. We first analyze the complexity of function PARETOSTEP realizing a single step on the path to the next local optimum.

**Lemma 13.** *Function* PARETOSTEP *executes in $O(n)$ time.*

*Proof.* We apply a constant number of mutations at each of the $O(n)$ plan nodes. We assume that plans are only pruned based on their cost values such that each instance of function PARETOSTEP returns only one non-dominated plan. Plan comparisons take $O(l) = O(1)$ time which leads to the postulated complexity. □

The expected time complexity of local search depends of course on the number of steps required to reach the next local Pareto optimum (a plan that is not dominated by any neighbor). We analyze the expected path length based on a simple statistical model in the following: we model the cost of a random plan according to one cost metric as random variable and assume that the random cost variables associated with different metrics are independent from each other. This assumption is simplifying but standard in the analysis of multi-objective query optimization algorithms [60].

**Lemma 14.** *The probability that a randomly selected plan dominates another is* $(1/2)^l$.

*Proof.* The probability that a randomly selected plan dominates another plan according to one single cost metric is $1/2$. Assuming independence between different cost metrics, the probability that a random plan dominates another one according to all $l$ cost metrics is $(1/2)^l$. □

**Lemma 15.** *The probability that none of n plans dominates all plans in a plan set of cardinality $i$ is* $(1-(1/2)^{li})^n$.

*Proof.* The probability that one plan dominates another is $(1/2)^l$ according to Lemma 14. The probability that one plan dominates all of $i$ other plans is $(1/2)^{li}$, assuming independence between the dominance probabilities between different plan pairs. The probability that a plan does not dominate all of $i$ other plans is $1-(1/2)^{li}$. The probability that none of $n$ plans dominates all of $i$ plans is $(1-(1/2)^{li})^n$. □

We denote by $u(n,i) = (1-(1/2)^{li})^n$ the probability that none of $n$ plans dominates all $i$ plans. We simplify in the following assuming that each plan has exactly $n$ neighbors.

**Theorem 21.** *The expected number of plans visited by the hill climbing algorithm until finding a local Pareto optimum is in* $\sum_{i=1..\infty} i \cdot u(n,i) \cdot \prod_{j=1..i-1}(1-u(n,j))$.

*Proof.* Our hill climbing algorithm visits a sequence of plans such that each plan is a neighbor of its successor and dominates its successor. Pareto dominance is a transitive relation and hence, as each plan dominates its immediate successor, each plan dominates all its successors. Then the probability of one additional step corresponds to the probability that at least one of the neighbors of the current plan dominates all plans encountered on the path so far. The probability that a local optimum is reached after $i$ plan nodes is the probability that none of the $n$ neighbors of the $i$-th plan dominates all $i$ plans on the path which is $u(n,i)$. The a-priori probability for visiting $i$ plans in total is then $u(n,i)\cdot\prod_{j=1..i-1}(1-u(n,j))$. The expected number of visited plans is $\sum_{i=1..\infty} i\cdot u(n,i)\cdot\prod_{j=1..i-1}(1-u(n,j))$. □

We bound the formula given by Theorem 21.

**Theorem 22.** *The expected path length from a random plan to the next local Pareto optimum is in $O(n)$.*

*Proof.* Assume that the hill climbing algorithm encounters at least $n$ plans on its path to the local Pareto optimum. We can write the expected number of additional steps as $\sum_{i=0..\infty} i\cdot u(n,n+i)\cdot\prod_{j=0..i-1}(1-u(n,j+n))$. Note that each additional step requires that one of the neighbors dominates at least $n$ plans on the path. Since $u(n,j)\leq 1$, we can upper-bound the additional steps by $\sum_{i=0..\infty} i\cdot\prod_{j=0..i-1}(1-u(n,j+n))$. Since $(1-u(n,j))$ is anti-monotone in $j$, we can upper-bound that expression by $\sum_{i=0..\infty} i\cdot(1-u(n,n))^i$. This expression contains the first derivative of the infinite geometric series and we obtain $(1-u(n,n))/(1-(1-u(n,n)))^2\leq 1/(1-(1-u(n,n)))^2=1/u(n,n)^2$ for the additional steps. We study how quickly that expression grows in $n$. It is $1/u(n,n)^2=1/(1-(1/2)^{ln})^{2n}\leq 1/(1-(1/2)^n)^{2n}\in O((1/(1-1/n)^n)^2)=O((1/e)^2)=O(1)$. The expected number of additional steps after $n$ steps is a constant. □

We finally calculated the expected time complexity of local search. Note that we make the pessimistic assumption that only one mutation is applied per path step while our algorithm allows in fact to apply many transformations together in one step.

**Theorem 23.** *Function* PARETOCLIMB *has expected time complexity in $O(n^2)$.*

*Proof.* We combine the expected path length (see Theorem 22) with the complexity per step (see Lemma 13). □

At this point it might be interesting to compare the overhead of hill climbing, calculated before, with the benefit it provides by reducing the search space. The factor by which the search space size is reduced when focusing on local optima is given by the following lemma.

**Lemma 16.** *The probability that a randomly selected plan is a local Pareto optimum is in $O((1-(1/2)^l)^n)$.*

*Proof.* A plan is a local Pareto optimum if none of its neighbors dominates the plan. The neighbor plans are created by applying a constant number of mutations at each plan node. For a plan joining $n$ tables, the number of neighbors is therefore in $O(n)$. We simplify by assuming that the probability that a plan is dominated by one of its neighbors corresponds to the probability that it is dominated by a random plan which is $(1/2)^l$ according to Lemma 14. The probability that a plan is not dominated by one of its neighbors is $1-(1/2)^l$ and the probability that it is not dominated by any of its neighbors is $O((1-(1/2)^l)^n)$. □

Multi-objective hill climbing leads to an exponential reduction of search space size while the expected path length to the next local Pareto optimum grows only linearly in the number of query tables. Next we analyze the complexity of generating a frontier approximation. We denote by $\alpha$ the precision factor that is used in the analyzed iteration. The next lemma is based on the proof of Lemma 2 in Chapter 2 bounding the number of query plans such that no plan approximately dominates another. Following their notation, we denote by $m$ the cardinality of the largest base table in the current database.

**Lemma 17.** *The plan cache associates $O((n \log_\alpha m)^{l-1})$ plans with a table set of cardinality $n$.*

*Proof.* New plans are only inserted into the plan cache if they are not approximately dominated by other plans joining the same tables, using approximation factor $\alpha$. The bound of $O((n \log_\alpha m)^{l-1})$ on the number of plans joining $n$ tables such that no plan approximately dominates another (see Chapter 2, Lemma 2) applies therefore to the plan cache. □

We denote by $b(n)$ the asymptotic bound on the number of plans stored per table set. Note that $b(n)$ grows monotonically in $n$.

**Theorem 24.** *Function* APPROXIMATEFRONTIER *has time complexity $O(n \cdot b(n)^3)$.*

*Proof.* The function treats each plan node in bottom-up order. For each node a set of new plans is generated and pruned by comparing it with alternative plans from the cache joining the same tables. We retrieve plans joining at most $n$ tables such that the number of plans retrieved for each table set is bounded by $b(n)$. Comparing one new plan against $b(n)$ stored plans is in $O(b(n))$. The complexity for treating inner nodes of the query plan tree (representing joins) is higher than the complexity of treating leaf nodes (representing scans). We generate $O(b(n)^2)$ new plans for an inner node which yields a per-node complexity of $O(b(n)^3)$ when taking pruning into account. Summing over all query plan nodes leads to the postulated complexity. □

The operation with dominant time complexity is the generation of the approximate frontier. This justifies that we start with a coarse-grained precision factor in order to explore a sufficient number of join orders quickly. The per-iteration complexity of RMQ follows immediately.

**Corollary 2.** *RMQ has time complexity $O(n \cdot b(n)^3)$ per iteration.*

We finally analyze the space complexity of RMQ. We analyze the accumulated space consumed after $i$ iterations. We assume that $b(n)$ designates the bound on the number of plans cached per table set for the precision factor $\alpha$ that is reached after $i$ iterations.

**Theorem 25.** *RMQ has space complexity $O(i \cdot n \cdot b(n))$.*

*Proof.* Each plan generates $O(n)$ intermediate results. We approximate the Pareto frontiers of all intermediate results that are used by one locally optimal plan in each iteration. Each iteration therefore adds at most $O(n)$ plan sets to the plan cache. The number of plans cached for each intermediate result is bounded by $b(n)$ and each plan requires $O(1)$ space as justified before. □

Considering multiple operator implementations changes time but not space complexity. We denote the number of implementations per operator by $r$. The asymptotic number of neighbor plans multiplies by $r$ and so does the expected path length to the nearest local optimum (this can be seen by substituting $n$ by $r \cdot n$ in Theorems 21 and 22). Hence the time complexity of local search multiplies by $r^2$. The time complexity of the frontier approximation multiplies by $r$ as we iterate over the operators in the innermost loop. The number of plan cost metrics is treated as constant as justified in Chapter 2.

## 5.6 Experimental Evaluation

We describe our experimental setup in Section 5.6.1 and discuss the results in Section 5.6.2.

### 5.6.1 Experimental Setup

We compare our RMQ algorithm against other algorithms for multi-objective query optimization. We compare algorithms in terms of how well they approximate the Pareto frontier for a given query after a certain amount of optimization time. We measure the approximation quality in regular intervals during optimization to compare algorithms in different time intervals. This allows to identify algorithms that quickly find reasonable solutions as well as algorithms that take longer to produce reasonable solutions but yield a better approximation in the end.

We judge the set of query plans produced by a certain algorithm by the lowest approximation factor $\alpha$ such that the produced plan set is an $\alpha$-approximate Pareto plan set. A lower $\alpha$ means a better approximation of the real Pareto frontier. This quality metric is equivalent to the $\varepsilon$ metric that was recommended in a seminal paper by Zitzler and Thiele [150] (setting $\alpha = 1+\varepsilon$). Choosing that metric also makes our comparison fair since the dynamic programming based approximation schemes from Chapter 2 against which we compare have been developed for that metric. As it is common in the area of multi-objective optimization, we often deal with test cases where finding the full set of Pareto solutions is computationally infeasible. We therefore compare the output of each algorithm against an approximation of the real Pareto frontier that is obtained by running all algorithms that we describe in the following for three seconds and taking the union of the obtained result plans.

We compare RMQ against two classes of algorithms: the dynamic programming based approximation schemes from Chapter 2 and generalizations of randomized algorithms that have been proposed for single-objective query optimization [131]. The approximation schemes guaran-

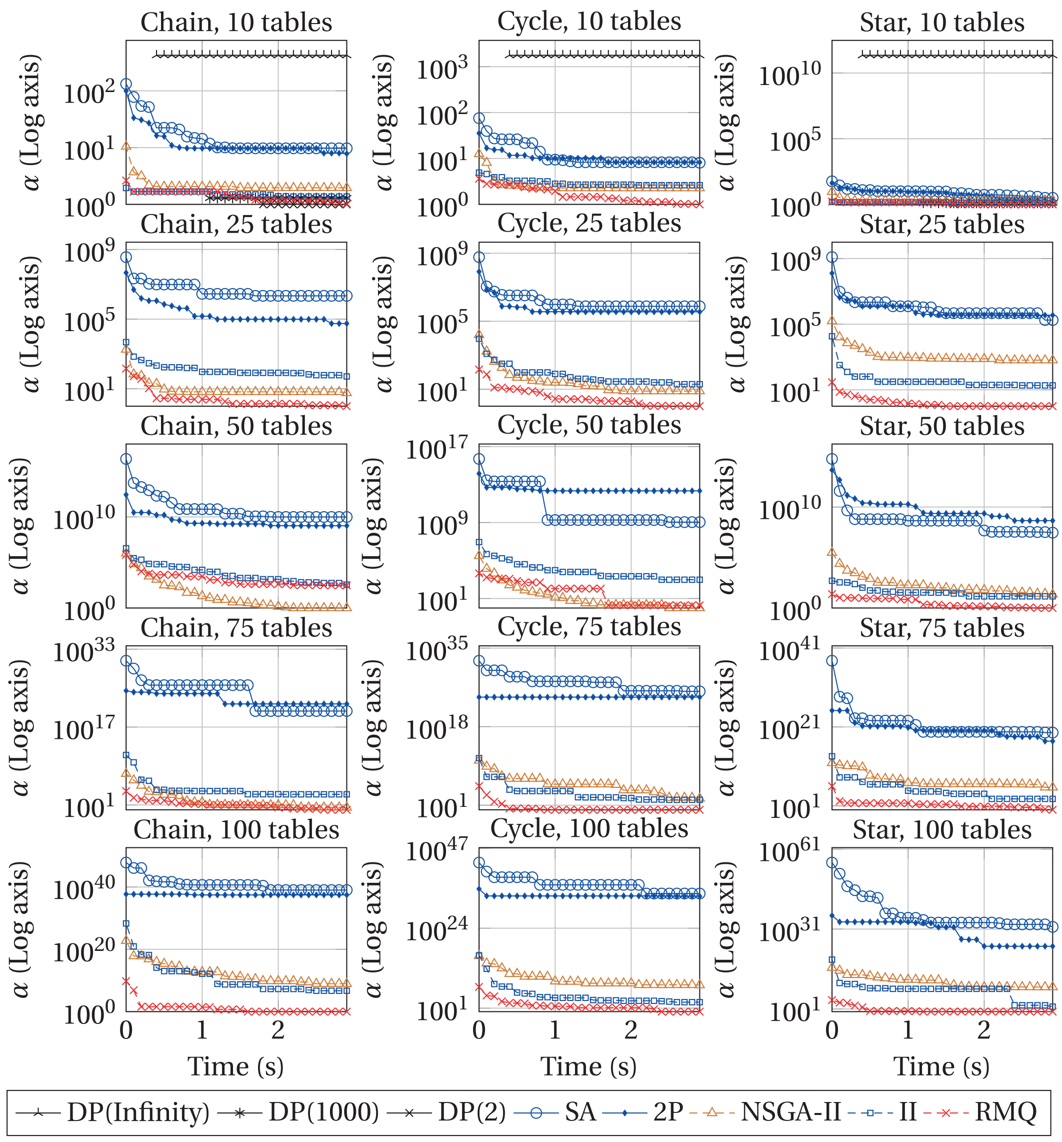


Figure 5.1 – Median of approximation error for two cost metrics as a function of optimization time.

tee to produce a Pareto frontier approximation whose $\alpha$ value does not exceed a user-specified threshold. We denote by DP($\alpha$) the approximation scheme with threshold $\alpha$. Choosing a higher value for $\alpha$ decreases optimization time but choosing a lower value leads to better result quality. We report results for different values of $\alpha$ in the following.

We experiment with four randomized algorithms (in addition to RMQ itself). By II we denote a generalization of iterative improvement [131] in which we iteratively walk towards local Pareto optima in the search space starting from random query plans. By SA we denote a generalization of the SAIO variant of simulated annealing, described by Steinbrunn et al. [131]. The original algorithm uses the difference between the scalar cost value of the current plan

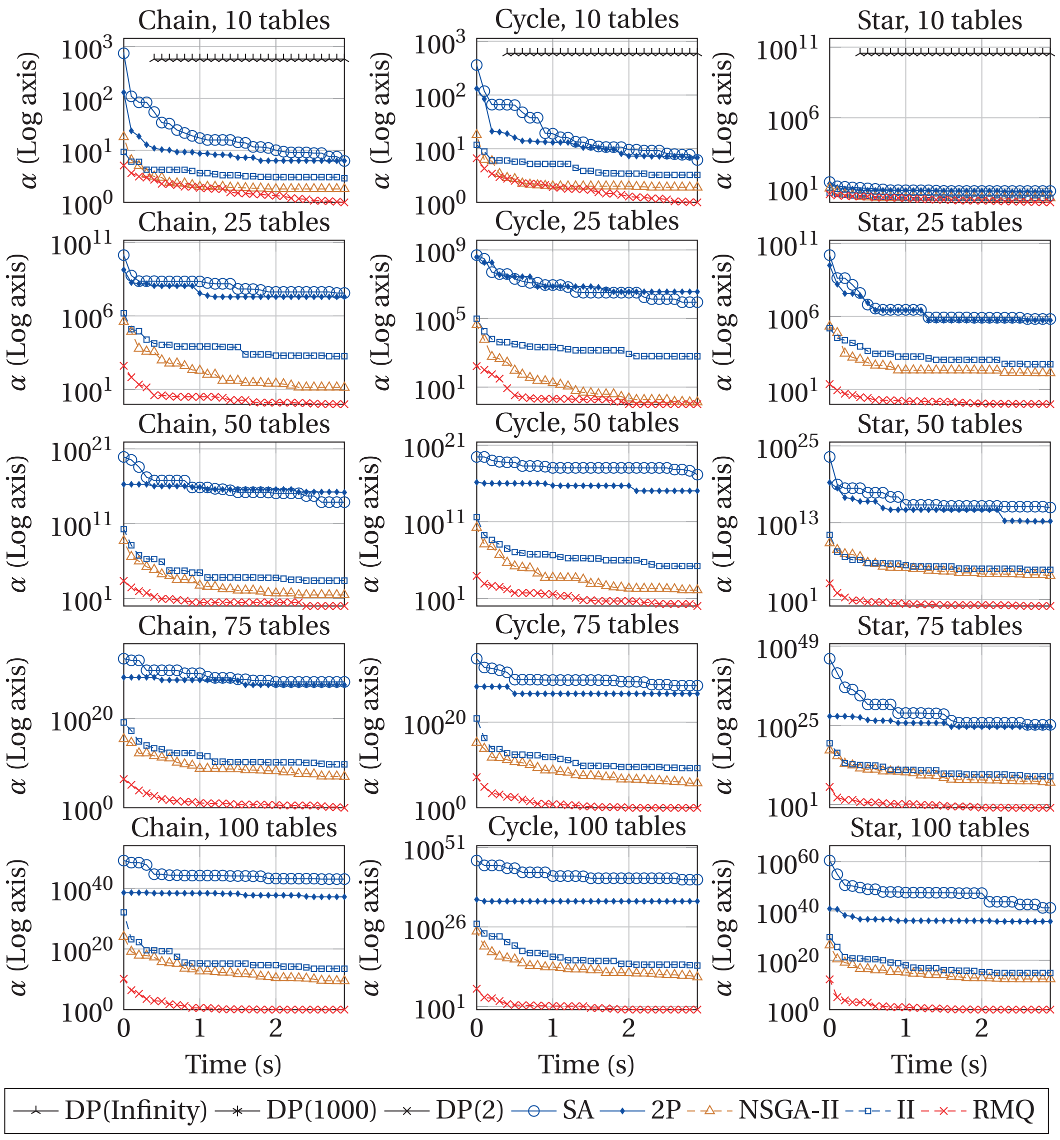


Figure 5.2 – Median of approximation error for three cost metrics as a function of optimization time.

and the cost of a randomly selected neighbor to decide whether to move towards the neighbor plan, based additionally on the current temperature. Our generalization uses the average cost difference between the current plan and its neighbor, averaging over all cost metrics. By 2P we abbreviate the two phase optimization algorithm for query optimization [131]. It executes the II algorithm in a first phase and continues with SA in a second phase. We switch to the second phase after ten iterations of II [131] and choose the initial temperature for SA as described by Steinbrunn et al. [131]. By NSGA-II we abbreviate the Non-Dominated Sort Genetic Algorithm II [49], a genetic algorithm for multi-objective query optimization that has been very successful for scenarios with the same number of cost metrics that we consider.

All our algorithms that are based on local search use the plan transformations for bushy query plans that were described by Steinbrunn et al. [131]. All algorithms using hill climbing (II and 2P) use the same efficient climbing function (see Algorithm 12) as our own algorithm. NSGA-II uses an ordinal plan encoding and a corresponding single-point crossover [131]. Our implementation of NSGA-II follows closely the pseudo-code given in the original paper [48] and we use the same settings for mutation and crossover probabilities. We use populations of size 200. We experimented with several configurations for each of the randomized algorithms (e.g., experimenting with different population sizes for NSGA-II and different number of examined neighbors for II, SA, and 2P) and report for each algorithm only the configuration that led to optimal performance.

We generate random queries with a given number of tables in the same way as in prior evaluations of query optimization algorithms [131, 31]: we experiment with different join graph structures and use stratified sampling to pick table cardinalities, using the same distribution as Steinbrunn et al. [131]. We consider up to three cost metrics on query plans that were already used for the experimental evaluations in the previous chapters: query execution time, buffer space consumption, and disc space consumption. We report in the following plots median values from 20 test cases per data point. For less than three cost metrics, we select the specified number of cost metrics with uniform distribution from the total set of metrics for each test case.

All algorithms were implemented in Java 1.7 and executed using the Java HotSpot(TM) 64-Bit Server Virtual Machine version on an iMac with i5-3470S 2.90GHz CPU and 16 GB of DDR3 RAM. We run each algorithm consecutively on all test cases after forcing garbage collection and after a ten seconds code warmup on randomly selected test cases in order to benchmark steady state performance[3].

### 5.6.2 Experimental Results

Figure 5.1 reports results for two cost metrics and Figure 5.2 reports our results for three cost metrics. We report separate results for different query sizes (measured by the number of tables being joined) and join graph structures (chain, cycle, and star shape). We allow up to three seconds of optimization time. The experiments for producing the data shown in Figures 5.1 and 5.2 took around eight hours of optimization time.

All algorithms presented in the previous chapters have so far been evaluated only on queries joining up to around 10 tables. For 10 table queries, DP(2) finds the best frontier approximation among all evaluated algorithms if two cost metrics are considered. For three cost metrics, DP(2) cannot finish optimization within the given time frame. For queries joining 25 tables and more, none of the approximation schemes finishes optimization within the given time frame. Note that the optimization time required by the approximation schemes we compare against constitutes a lower bound for the optimization time required by the incremental

[3]http://www.ibm.com/developerworks/library/j-benchmark1/

approximation algorithm described in Chapter 3 for reaching the same approximation quality. As the non-incremental approximation schemes cannot finish optimization within the given time frame, even when setting $\alpha = \infty$, the incremental versions will not be able to do so either. This is not surprising as we reach a query size where randomized algorithms would already be used for single-objective query optimization which is computationally far less expensive than multi-objective query optimization. This shows the need for randomized algorithms for multi-objective query optimization such as RMQ.

Among the randomized algorithms that generalize popular randomized algorithms for single-objective query optimization (II, SA, 2P, and NSGA-II), II and NSGA-II generate the best approximations with a large gap to SA and 2P (note the logarithmic y axis for approximation error). It is interesting that II outperforms 2P unlike in traditional query optimization where the roles are reversed. However, SA and 2P both spend most of their time improving one single query plan (2P after limited initial sampling) and are therefore intrinsically based on the assumption that only one very good plan needs to be found as result. Approximating the Pareto frontier requires however to generate a diverse set of query plans which is better accomplished by the seemingly naive II which starts each iteration with a new random plan. NSGA-II usually performs better than II, SA, and 2P. This is not surprising since NSGA-II is a popular algorithm for multi-objective optimization with a moderate number of cost metrics and genetic algorithms have been shown to perform well for classical query optimization [23].

Our own algorithm, RMQ, outperforms all other algorithms in the majority of cases. The gap to the other algorithms increases in the number of query tables and in the number of cost metrics. For two cost metrics (see Figure 5.1), RMQ is competitive and often significantly better than all other algorithms over the entire optimization time period starting from more than 50 join tables. For star-shaped query graphs, RMQ is better than competing algorithms starting from 25 join tables already. Considering three cost metrics (see Figure 5.2) increases the gap between RMQ and all other algorithms. Starting from 25 join tables, RMQ dominates over the entire optimization time period. The gap in terms of approximation error reaches many orders of magnitude for large queries.

Figures 5.3 shows additional statistics: on the left side we see that the average path length from a random plan to the nearest Pareto optimum (using function ParetoClimb) grows slowly in the number of query tables as postulated in our formal analysis. On the right side, we see that the number of Pareto plans grows in the number of query tables which corroborates the results from Chapter 2. This tendency explains why the approximation error of randomized algorithms increases in the query size: having more Pareto plans makes it harder to obtain a good approximation.

We summarize the main results of our experimental evaluation. Dynamic programming based algorithms for multi-objective query optimization are only applicable for small queries. Using randomized algorithms, we were able to approximate the Pareto frontier for large queries joining up to 100 tables. The algorithm proposed in this chapter performs best

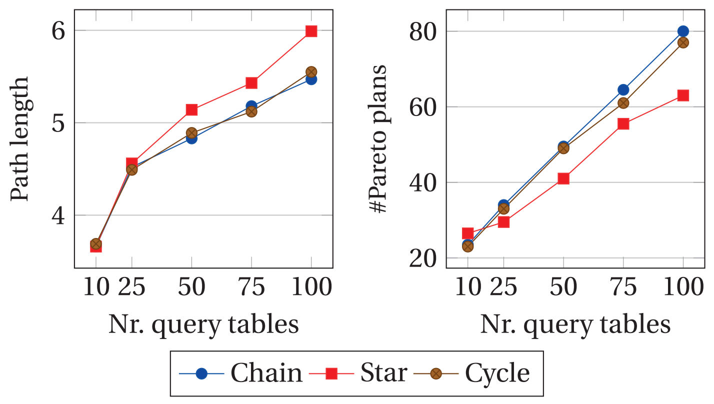


Figure 5.3 – Median path length from random plan to next local Pareto optimum and median number of Pareto plans found by RMQ for three cost metrics.

among the randomized algorithms over a broad range of scenarios. Among the remaining randomized algorithms, a general-purpose genetic algorithm for multi-objective optimization performs best, followed closely by an iterative improvement algorithm that uses the efficient hill climbing function that was introduced in this chapter as well.

So far we have compared algorithms for a few seconds of optimization time. It is generally advantageous to minimize optimization time as it adds to execution time. In the context of multi-objective query optimization, it is even more important as query optimization can be an interactive process in which users have to wait until optimization finishes (see Chapter 3). We have however also compared algorithms for up to 30 seconds of optimization time and present the corresponding results in the next section. Until now we compared algorithms in terms of how well they approximate an approximated Pareto frontier since calculating the real Pareto frontier would lead to prohibitive computational costs for many of the query sizes we consider. We show results for small queries where we calculated the real Pareto frontier in the next section. Furthermore, the next section contains results for different query generation methods. The main results of our experiments are stable across all scenarios.

## 5.7 Additional Experimental Results

The performance of query optimization algorithms may depend on the probability distribution over predicates selectivity values used during random query generation. For our experiments in Section 5.6, we used the original method proposed by Steinbrunn et al. [131] to select the selectivity of join predicates. We performed an additional series of experiments in which we select predicate selectivity according to the MinMax method proposed by Bruno instead [31]. Using that method, each join has an output cardinality between the cardinalities of the two input relations. The purpose of those additional experiments is to verify whether our experimental results from Section 5.6 generalize. We report the results of the second series of experiments in Figures 5.4 and 5.5. The experimental setup is the same as described in

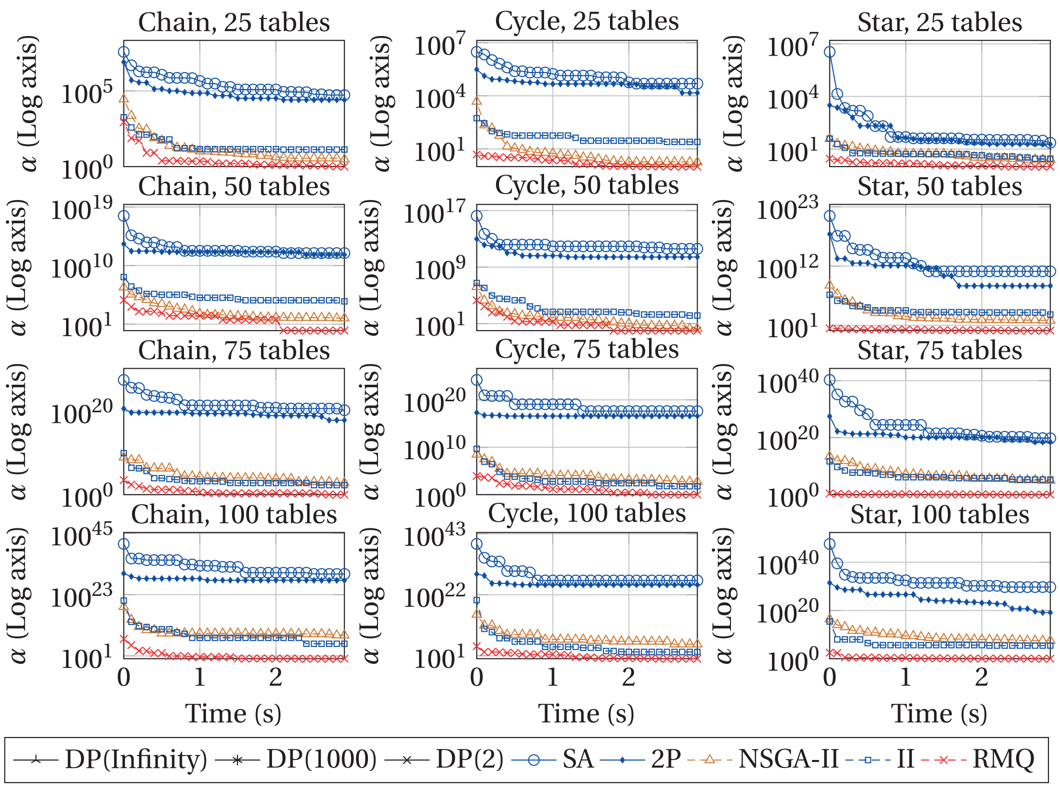


Figure 5.4 – Median of approximation error for two cost metrics as a function of optimization time (MinMax joins).

Section 5.6.1 except for the choice of selectivity values. We focus on queries joining between 25 to 100 tables where randomized algorithms become better than dynamic programming variants.

The results are largely consistent with the results we obtained in our first series of experiments. Our own algorithm outperforms all other approaches significantly for large queries and many cost metrics, in particular during the first second of optimization time. NSGA-II performs well for smaller queries while its approximation error is by many orders of magnitude sub-optimal for large queries. II comes close to NSGA-II in certain scenarios while the two randomized algorithms based on simulated annealing, SA and 2P, perform badly. The approximation schemes do not scale to queries of 25 tables and more.

Multi-objective query optimization needs to integrate user preferences in order to determine the optimal query plan. One possibility to integrate user preferences is to present an approximation of the plan Pareto frontier for a given query to the user such that the user can select the preferred cost tradeoff (see Chapter 3). This means that users have to wait after submitting a query until optimization finishes in order to make their selection. In that scenario, an optimization time of only a few seconds is desirable. If users formalize their preferences before optimization starts (e.g., in the form of cost weights and cost bounds as in Chapter 2) then

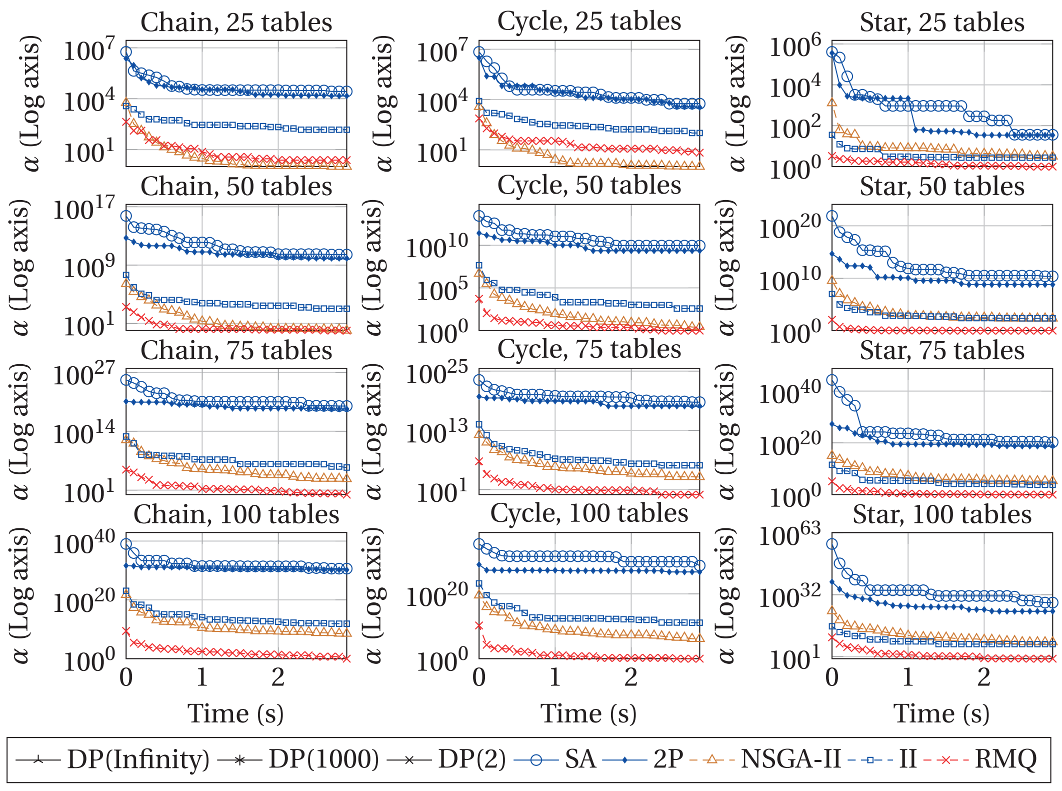


Figure 5.5 – Median of approximation error for three cost metrics as a function of optimization time (MinMax joins).

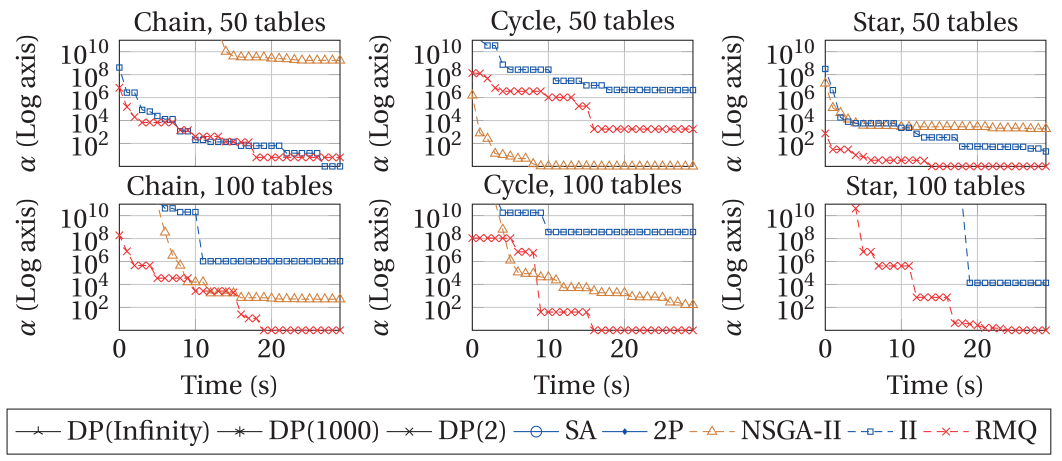


Figure 5.6 – Median of approximation error in the interval $[1, 10^{10}]$ for two cost metrics and up to 30 seconds of optimization time.

longer optimization times can be acceptable. The second scenario motivates us to compare the query optimization algorithms for longer periods of optimization time.

We executed another series of experiments giving each algorithm 30 seconds of optimization

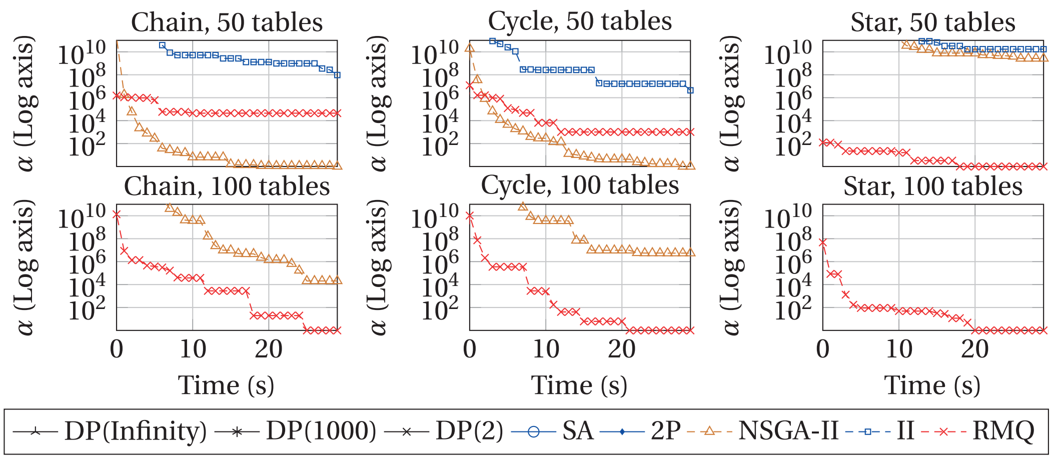


Figure 5.7 – Median of approximation error in the interval $[1, 10^{10}]$ for three cost metrics and up to 30 seconds of optimization time.

time. We reduced the number of test cases per scenario from 20 to 10 and only experimented with two query sizes in order to decrease the time overhead for the experiments. Figures 5.6 and 5.7 report the development of the median approximation error over 30 seconds of optimization time. We restrict the y domain of the figures and only show errors up to $\alpha = 10^{10}$. Using that thresholds allows us to visualize performance difference between the well performing algorithms that would otherwise become indistinguishable even though they are significant. Albeit we ran the same algorithms as in the previous plots on all test cases, the plots only show data points for a subset of those algorithms. For the dynamic programming variants, the reason for not being represented in the plots is that they did not return any results even within 30 seconds of optimization time. For simulated annealing and two-phase optimization, the reason for not being represented in the plots is that their approximation error is significantly above the threshold of $10^{10}$ (often more than $10^{110}$).

The tendencies remain the same: our randomized algorithm, the genetic algorithm, and iterative improvement with our fast climbing function are the best randomized algorithms. RMQ is usually better than iterative improvement. For queries with up to 50 tables, it depends on the number of cost metrics and the join graph structure whether RMQ or the genetic algorithm perform better. For more than 50 tables, RMQ outperforms all other algorithms over most of the optimization time period, the margin increases in the number of plan cost metrics.

We have finally run an additional test series in which we calculate the approximation error more precisely than in the previous experiments. For large queries, we have no choice but to evaluate approximation precision with regards to an approximated Pareto frontier that is generated by the same randomized algorithms that we evaluate. For small queries, we can use the dynamic programming based approximation schemes to calculate an approximated Pareto frontier with formal guarantees on how closely it is approximated. For the last series of

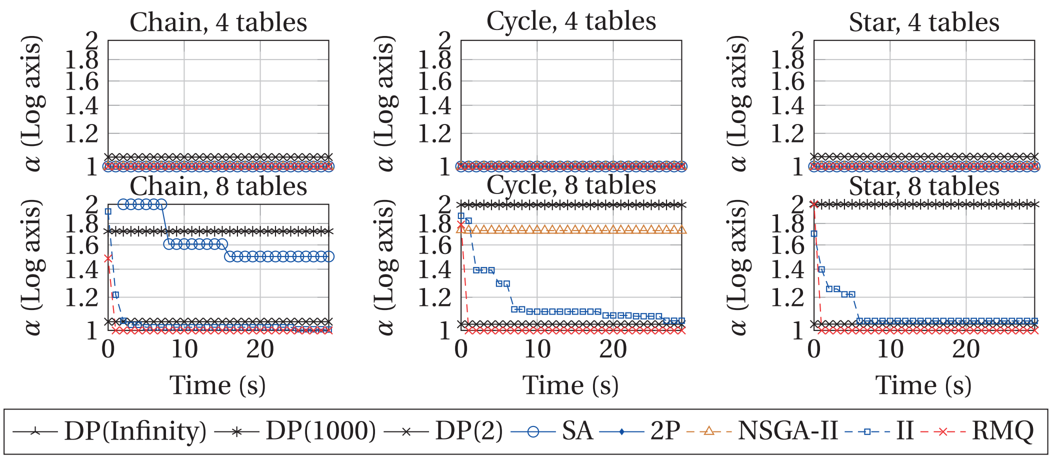


Figure 5.8 – Median of precise approximation error in the interval [1,2] for small queries and two cost metrics.

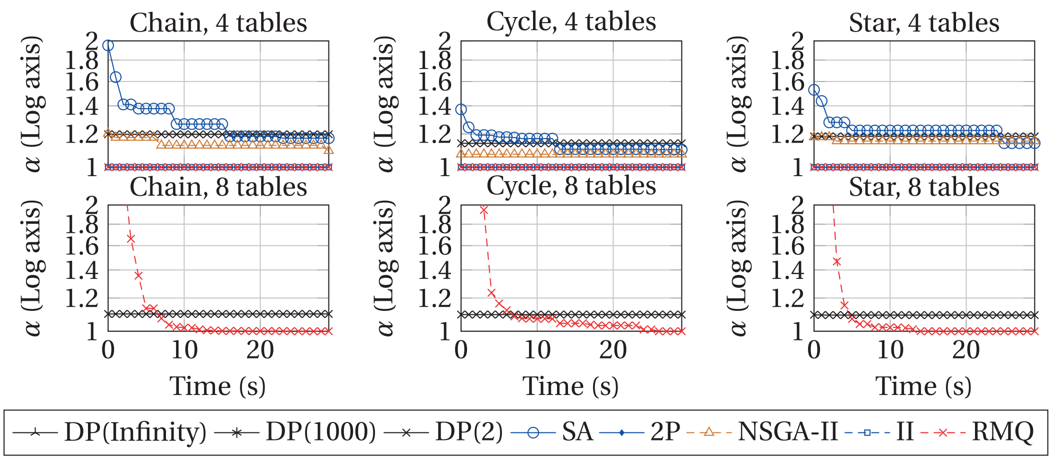


Figure 5.9 – Median of precise approximation error in the interval [1,2] for small queries and three cost metrics.

experiments, we calculate the approximated Pareto frontier by the approximation scheme, setting $\alpha = 1.01$. This means that the calculated approximation error is guaranteed to be precise within a very small tolerance. We restrict ourselves to small queries joining between four and eight tables. Note that the size of the Pareto frontier produced by the approximation scheme reaches several hundreds of Pareto plans already for such small queries.

Figures 5.8 and 5.9 show the results. We allow again 30 seconds of optimization time and restrict the plots to the domain [1,2] for the approximation error. This has the same implications as discussed in the previous paragraphs: algorithms with exceedingly high errors are not shown in order not to obfuscate the performance differences between the competitive algorithms. We are primarily interested in whether or not the randomized algorithms converge to the reference Pareto frontier for small queries; therefore we choose to show a small error range above $\alpha = 1$.

We find that our algorithms converges in average to a perfect approximation with $\alpha = 1$ while this is not always the case for the other algorithms. In particular for queries joining eight tables and three plan cost metrics, our algorithm is the only one among all randomized algorithms that achieves a perfect approximation. The dynamic programming based approximation scheme with $\alpha = 2$ performs well for small queries. It generates output nearly immediately and the approximation error is much lower than the theoretical worst case bound. The approximation scheme configuration using $\alpha = 1.01$, the one generating the reference frontier, produces its solutions after less than two seconds of optimization time in average, even for three cost metrics and eight tables. We do not show results for that algorithm in the plots as its approximation error is minimal by definition. We conclude that approximation schemes often outperform randomized algorithms for small queries.

## 5.8 Conclusion

The applicability of multi-objective query optimization has so far been severely restricted by the fact that all available algorithms have exponential complexity in the number of query tables. We presented, analyzed, and evaluated the first polynomial time heuristic for multi-objective query optimization. We have shown that our algorithm scales to queries that are by one order of magnitude larger than the ones prior multi-objective query optimizers were so far evaluated on. We envision our algorithm being used in future multi-objective query optimizers that apply dynamic programming based algorithms for small queries and switch to randomized algorithms starting from a certain number of query tables, similar to what single-objective optimizers do today.

```
// Plan $p_1$ is better than $p_2$ if it produces the same
// data format as $p_2$ and has dominant cost.
function BETTER($p_1, p_2$)
    return SAMEOUTPUT($p_1, p_2$)$\wedge$($p_1 \prec p_2$)
end function
// Keeps one Pareto plan per output format.
function PRUNE($plans, newPlan$)
    if $\nexists p \in plans$:BETTER($p, newPlan$) then
        $plans \leftarrow \{p \in plans | \neg$BETTER$(newPlan, p)\}$
        $plans \leftarrow plans \cup \{newPlan\}$
    end if
    return $plans$
end function
// Improve plan $p$ by parallel local transformations
function PARETOSTEP($p$)
    // Initialize optimal mutations of this plan
    $pPareto \leftarrow \emptyset$
    if $p.isJoin$ then
        // Improve sub-plans by recursive calls
        $outerPareto \leftarrow$PARETOSTEP($p.outer$)
        $innerPareto \leftarrow$PARETOSTEP($p.inner$)
        // Iterate over all improved sub-plan pairs
        for $outer \in outerPareto$ do
            for $inner \in innerPareto$ do
                $p.outer \leftarrow outer$
                $p.inner \leftarrow inner$
                // Mutations for specific sub-plan pair
                for $mutated \in$MUTATIONS($p$) do
                    $pPareto \leftarrow$PRUNE($pPareto, mutated$)
                end for
            end for
        end for
    else
        // $p$ is single-table scan
        for $mutated \in$MUTATIONS($p$) do
            $pPareto \leftarrow$PRUNE($pPareto, mutated$)
        end for
    end if
    return $pPareto$
end function
// Climbs until plan $p$ cannot be improved further.
function PARETOCLIMB($p$)
    $improving \leftarrow$true
    while $improving$ do
        $improving \leftarrow$false
        $mutations \leftarrow$PARETOSTEP($p$)
        if $pm \in mutations : pm \prec p$ then
            $p \leftarrow pm$
            $improving \leftarrow$true
        end if
    end while
    return $p$
end function
```

Algorithm 12 – Fast multi-objective hill climbing performing mutations in independent plan sub-trees simultaneously.

```
// Checks if plan $p_1$ is significantly better than $p_2$
// using coarsening factor $\alpha$ for cost comparison.
**function** SIGBETTER($p_1, p_2, \alpha$)
    **return** SAMEOUTPUT($p_1, p_2$)$\wedge p_1 \preceq_\alpha p_2$
**end function**
// Returns an $\alpha$-approximate Pareto frontier
**function** $Prune(plans, newP, \alpha)$
    // Can we approximate the cost of the new plan?
    **if** $\nexists p \in plans$:SIGBETTER($p, newP, \alpha$) **then**
        $plans \leftarrow \{p \in plans | \neg$SIGBETTER$(newP, p, 1)\}$
        $plans \leftarrow plans \cup \{newP\}$
    **end if**
    **return** $plans$
**end function**
// Approximates the Pareto frontier for each
// intermediate result that appears in plan $p$,
// using partial plans from the plan cache $P$.
// The precision depends on the iteration count $i$.
**function** APPROXIMATEFRONTIERS($p, P, i$)
    // Calculate target approximation precision
    $\alpha \leftarrow 25 \cdot 0.99^{\lfloor i/25 \rfloor}$
    **if** $p.isJoin$ **then**
        // Approximate outer and inner plan frontiers
        $P \leftarrow$APPROXIMATEFRONTIERS($p.outer, P, i$)
        $P \leftarrow$APPROXIMATEFRONTIERS($p.inner, P, i$)
        // Iterate over outer Pareto plans
        **for** $outer \leftarrow P[p.outer.rel]$ **do**
            // Iterate over inner Pareto plans
            **for** $inner \leftarrow P[p.inner.rel]$ **do**
                // Iterate over applicable join operators
                **for** $op \in$JOINOPS($outer, inner$) **do**
                    // Generate new plan and prune
                    $np \leftarrow$JOINPLAN($outer, inner, op$)
                    $P[p.rel] \leftarrow$PRUNE($P[p.rel], np, \alpha$)
                **end for**
            **end for**
        **end for**
    **else**
        // Iterate over applicable scan operators
        **for** $op \in$SCANOPS($p.rel$) **do**
            $np \leftarrow$SCANPLAN($p.rel, op$)
            $P[p.rel] \leftarrow$PRUNE($P[p.rel], np, \alpha$)
        **end for**
    **end if**
    // Return updated plan cache
    **return** $P$
**end function**
```

Algorithm 13 – Approximating the Pareto frontiers of all intermediate results occuring within given plan.

# 6 Massive Parallelization

Randomization is a technique that enables us to optimize even large queries with multiple cost metrics. The drawback of randomization is that we give up any formal worst-case guarantees on the quality of the resulting query plans. In this chapter, we explore an alternative technique that allows optimizing large queries while maintaining formal guarantees: massive parallelization. We will see a decomposition method that allows parallelizing query optimization over large clusters with hundreds of nodes in a shared-nothing architecture. This decomposition method works for classical query optimization, for multi-objective query optimization, for parametric query optimization, and for multi-objective parametric query optimization. Prior approaches for parallelizing query optimization have aimed at moderate degrees of parallelism in shared-memory architectures. We will see that exploiting massive degrees of parallelism requires a very different decomposition approach.

## 6.1 Introduction

Moore's law [102] is breaking and computer systems become more powerful by increasing their number of processing units (be it cores, CPUs, or cluster nodes) rather than by increasing clock rates. This means that all stages of query evaluation must exploit parallelism in order not to become the bottleneck in future systems.

Research on parallelizing query evaluation has so far mainly focused on how to parallelize the actual query processing stage, i.e. how to parallelize the execution of query plans. This is however insufficient as noted in prior work [67, 68, 145, 129]: in order to parallelize query evaluation, we must not only parallelize the execution of query plans but also the *generation* of query plans, i.e. we must develop parallel algorithms for the query optimization problem.

Query optimization is an NP-hard problem and even finding guaranteed near-optimal query plans is NP-hard [33]. The run time of all known algorithms increases exponentially in the number of joins and novel application scenarios (e.g., SPARQL query processing [45]) motivate queries with many joins. Furthermore, the complexity of the systems on which query processing takes place increases: the number of system components keeps increasing (as discussed before), flexible provisioning models and novel processing operators introduce new parameters by which query processing can be tuned (e.g., the number of machines to rent is such a parameter in a cloud scenario [88] or the sampling rate of a scan operator in the context of approximate query processing [12]). All those developments make query optimization harder since the size of the plan search space increases. In addition, many of the aforementioned developments motivate new cost metrics for comparing query plans (e.g., monetary fees in a cloud scenario or result precision in approximate query processing) in addition to execution time. Having multiple plan cost metrics makes query optimization however harder as demonstrated in the previous chapters. In summary, there are many ongoing developments that make query optimization harder and hence increase the need for parallel query optimization algorithms.

We propose a novel, parallel algorithm for query optimization in this work. Our goal is to obtain a query optimization algorithm that is future-proof in that it is able to exploit the ever-growing degree of parallelism forced by the breakdown of Moore's law. While prior parallel query optimization algorithms have been primarily designed for shared-memory architectures, we aim at parallelizing query optimization on shared-nothing architectures as well. Query plans are often executed on large clusters and, as query optimization must precede query execution, it is preferable to use all cluster nodes for query optimization rather than leave them idle until optimization has finished. Even for queries that are executed repeatedly on a single node, a cluster can be used for optimization before run time if optimization is expensive. The algorithm that we propose is however not specific to shared-nothing architectures and can be applied in different scenarios as well.

Prior approaches for parallelizing query optimization assume that worker threads share common data structures [67, 68, 145, 37, 129], in particular big memotables storing subsets of query tables optimal join plans. They assume that a central master node distributes fine-grained optimization tasks to workers and that many interactions between master and worker threads take place during the optimization of a single query. In a shared-nothing architecture, sharing data between worker threads results in high communication overhead and each task assignment incurs setup overhead. We target extremely high degrees of parallelism, at least several hundreds of cluster nodes (while prior algorithms have not been evaluated on more than eight cores). Orchestrating that many nodes on the level of micro optimization tasks results in prohibitive communication and computation overhead on the master node.

Achieving our goals requires a radically different approach compared to prior work: instead of decomposing the query optimization problem into many small optimization tasks, we realize the most coarse-grained problem decomposition possible: the optimization of one query is mapped into exactly one task per worker node.

On a high level, our algorithm works as follows. Given a query to find an optimal plan for, the master optimizer node sends that query together with a plan space partition ID to each worker node. The partition ID is simply an integer between one and the number of workers such that each worker obtains a different number. Each worker node translates its partition ID into a set of constraints on join orders and only considers query plans that comply with those constraints. Each worker node therefore searches for an optimal plan within a plan space that is smaller than the original plan space. The worker nodes search the optimal plan within their respective plan space partition in parallel. No communication between workers or between workers and master node is required during that stage. Afterwards, the workers send the optimal plans back to the master node. The original plan space is the union over all plan space partitions. Comparing the plans returned by the workers, which are optimal within their respective partition and whose number is linear in the number of workers, yields therefore the globally optimal plan.

Our algorithm is designed to exploit very high degrees of parallelism. The time complexity of all serial processing steps, executed by the master node, is linear in the number of workers and in the query size. The amount of data sent over the network is also linear in the number of workers and in the query size. All plan space partitions have the same size which guarantees skew-free parallelization. For a fixed query, the run time as well as the consumption of main memory space per worker node decreases monotonically in the number of worker nodes. Furthermore, the number of partitions into which the plan space can be divided and therefore the maximal degree of parallelism grows in the query size and is in principle unlimited.

Our algorithm parallelizes one of the most popular dynamic programming schemes for query optimization [116]. It treats table sets of increasing cardinality and constructs optimal join plans for each table set out of optimal plans for table subsets that were previously generated. As it has been noted in prior work [67], this dynamic programming scheme belongs to the class of non-serial polyadic algorithms and is therefore difficult to parallelize. Certainly it is easier to parallelize randomized query optimization algorithms such as iterated improvement or simulated annealing [134, 77]. We nevertheless focus on parallelizing the dynamic programming approach. There are two reasons. First, unlike randomized algorithms, the dynamic programming approach formally guarantees to return optimal query plans. Second, by parallelizing Sellinger's classical dynamic programming scheme [116] we parallelize at the same time many query optimization algorithms that have been based on the same scheme and cover a multitude of scenarios (e.g., multi-objective query optimization and multi-objective parametric query optimization, the variants discussed in the previous chapters, or parametric query optimization [74]).

The time and space complexity of the classical dynamic programming algorithm depend on the number of table sets for which optimal join plans need to be found. We decompose the query optimization problem by introducing constraints on the join order that ultimately allow to reduce the number of table sets to consider.

We propose a partitioning scheme for the space of left-deep query plans and one partitioning method for bushy query plans. Left-deep query plans are characterized by the order in which tables are joined. We restrict join orders by constraints of the form $x \prec y$ where $x$ and $y$ are query tables: the semantics is that table $x$ needs to be joined before table $y$. The constraint excludes any query plan producing an intermediate join result containing table $y$ but not table $x$ and hence we can neglect table sets containing $y$ without $x$ during dynamic programming. This reduces the number of table sets to consider by a factor of $3/4$. If we assign the constraint $x \prec y$ to a first worker node and the complementary constraint $y \prec x$ to a second worker then the entire search space is covered. Furthermore, we can recursively decompose the resulting plan space partitions by applying similar constraints to other (disjoint) table pairs.

Bushy query plans are binary trees and cannot be represented as join orders anymore. However, if we fix an arbitrary table and follow its way from a leaf node in the plan tree to the root then we can order the other tables based on when they first appear in the sequence of intermediate results we encounter. Hence we restrict join orders for bushy plan spaces by constraints of the form $x \preceq y|z$ with the semantics that $x$ appears no later than $y$ when following table $z$ to the plan tree root. This excludes join results that contain tables $y$ and $z$ but not table $x$.

We formally analyze time and space complexity and the network bandwidth required by our algorithm. We show that each constraint reduces time complexity by factor $3/4$ for linear and by factor $21/27$ for bushy plan spaces. We show that those reduction factors are actually optimal within a restricted design space of partitioning methods. Prior algorithms achieved near linear speedups until a low number of threads within a shared-memory architecture. Our speedups are not linear but very steady up to very high degrees of parallelism and within a shared-nothing architecture. In our experiments, we demonstrate continuous scaling up to more than 250 concurrent worker threads on a large cluster over various query sizes and for single as well as multi-objective query optimization. As our algorithm scales even in this challenging scenario, we believe that it scales on many other architectures as well.

The original scientific contributions of this chapter are in summary the following:

- We propose a novel algorithm for massively-parallel query optimization on shared-nothing architectures.
- We formally evaluate that algorithm in terms of time and space complexity and in terms of the required network traffic.
- We evaluate the algorithm experimentally on a large cluster, demonstrating its scalability for up to more than 250 concurrent worker threads.

The remainder of this chapter is organized as follows. We compare against related work in Section 6.2. In Section 6.3, we introduce our formal problem model. We present our algorithms for parallel query optimization in left-deep and bushy plan spaces in Section 6.4. In Section 6.5, we analyze time and space complexity as well as the growth in network traffic. In Section 6.6, we experimentally demonstrate the scalability of our algorithms on a large cluster.

## 6.2 Related Work

The term *parallel query optimization* sometimes refers to serial optimization algorithms generating plans that are executed in parallel [36]. It is crucial to realize that we use the term in a very different sense: we propose a parallel algorithm for generating query plans (that may be executed serially or in parallel).

Our work connects to prior work that parallelizes the classical dynamic programming based query optimization algorithm [67, 68, 145, 151, 37, 129]. Prior algorithms have however implicitly been designed for shared-memory architectures that do not scale beyond a certain degree of parallelism [132]. Prior algorithms have been evaluated on up to maximally eight cores while we demonstrate scalability of our algorithm on a shared-nothing architecture using over 250 workers. We outline some of the factors that distinguish prior algorithm from our algorithm and limit their scalability.

Prior algorithms assume that all threads share common data structures (e.g., the memotable containing optimal partial plans) and hence can access intermediate results generated by other threads. This would lead to huge communication overhead on shared-nothing architectures (e.g., the size of the memotable is exponential in the query size) while our algorithm does not require any communication between workers. Furthermore, prior algorithms use a central coordinator which assigns rather fine-grained optimization tasks to worker threads (e.g., the master thread assigns specific pairs of join operands to generate plans for). This has two disadvantages. First, a lot of communication is required between master and workers. Second, the time complexity for managing the workers is high, so the master itself will eventually become the bottleneck as the degree of parallelism increases.

We assign tasks at the coarsest possible level: each worker receives exactly one task per query. The time complexity of the algorithm executed on the master is linear in the number of worker nodes and in the query size and so is the total amount of data that needs to be sent over the network. Finally, only one round of communication between workers and master is required per query by our algorithm while prior algorithms usually require many rounds of communication. Having only one round of communication is advantageous in scenarios where distributing tasks to workers and receiving the results is associated with overheads. We compare against a typical representative of prior algorithms in our experimental evaluation.

Our work is generally relevant for all areas of query optimization in which algorithms based on dynamic programming have been proposed. This includes, for instance, multi-objective query optimization and multi-objective parametric query optimization, the variants discussed in the previous chapters, and parametric query optimization [59, 78]. Our method of partitioning the join order space is generic and can be applied to all of those scenarios.

## 6.3 Problem Model

Our notation is similar to the ones used in previous chapters. Nevertheless, we introduce notation from scratch to make the current chapter self-contained.

As it is standard in query optimization, we use a simplified query and query plan model to describe our algorithms. Extending the model and the algorithms towards richer query languages and plan spaces is however straightforward and can be achieved via standard techniques [116].

A query is a set $Q$ of tables that need to be joined. We denote by $\text{SCAN}(q)$ for $q \in Q$ a query plan that scans a single table and call such a plan a *scan plan*. By $\text{JOIN}(p_L, p_R)$ we designate a plan that joins the result produced by plan $p_L$ with the result produced by $p_R$ and uses $p_L$ as outer and $p_R$ as inner operand. We use the terms left and right operand as synonyms for outer and inner operand respectively as the outer operand is usually drawn at the left side in visual representations of query plans. Note that we do not incorporate alternative operator implementations for scans and joins into our model to simplify the presented pseudo-code. The extension is however easy and the implementation of our algorithm used for the experiments considers all standard operators.

We distinguish two types of query plans. *Left-deep plans* are plans in which the right operand of every join is a scan plan. All other plans are *bushy plans*. Bushy plans can be represented as labeled binary trees where leaf nodes correspond to single tables and inner nodes correspond to join results. The tree shape of left-deep plans is fixed and the join order of a left-deep plan is fully described by the order in which table leaf nodes are encountered in a traversal (e.g., in post-order) of the plan tree. This is why we can represent left-deep plans by a sequence of query tables.

For a fixed query, the set of all bushy plans is the *bushy plan space* and the set of all left-deep plans is the *left-deep* or *linear plan space*. We assume that a cost model is available that associates query plans with cost estimates. Our pseudo-code encapsulates that cost model in a pruning function that discards the plan with higher cost among several compared plans. The goal of query optimization is to find the cost-optimal plan either in the space of left-deep or in the space of bushy plans.

## 6.4 Algorithm

We present an algorithm for massively-parallel query optimization. The algorithm is well suited for shared-nothing architectures as it minimizes the amount of sychronization and communication overhead. The same properties are however beneficial in shared-memory scenarios. Our algorithm is not specific to shared-nothing architectures and can be used to parallelize query optimization over the nodes of a cluster or over the cores of a single computer all the same.

The presented algorithm solves the traditional query optimization problem, meaning that it compares alternative query plans according to single point cost estimates in one cost metric. The method by which we partition the plan space is however very generic and it is in fact straight-forward to extend our algorithm to handle multiple plan cost metrics (as in the previous chapters) or plan cost functions that depend on unknown parameters [78, 59] or both together (as in Chapter 4). This is possible since algorithms have been proposed for all of the aforementioned query optimization variants that use the same dynamic programming scheme as the classical algorithm by Selinger [116]; only the pruning function, the way in which different query plans are compared, differs between them. The algorithm presented next can therefore easily be transformed into an algorithm handling other query optimization variants by essentially replacing the pruning function.

We present two variants of our algorithm: the first variant finds the optimal left-deep query plan for a given query while the second variant finds the optimal plan within a bushy plan space. Before discussing the pseudo-code, we illustrate informally how our algorithm works by means of a simplified example. This example refers to the algorithm variant searching left-deep plan spaces.

**Example 13.** *Assume we want to find the optimal left-deep plan for answering the join query* $R \bowtie S \bowtie T \bowtie U$*. Further assume that four worker nodes are available over which query optimization is parallelized. Upon reception of the query, the master nodes sends the query together with the total number of plan space partitions (four) and the respective partition ID (between one and four) to each worker node. Consider the worker node that partition three is assigned to. Knowing that the total number of partitions is four, the worker node derives that it should use* $\log_2 4 = 2$ *constraints to restrict the join order space. The two constraints refer to the order in which the four tables are joined. The first constraint refers to the ordering between the first pair of tables,* $R$ *and* $S$*, and establishes which of them appears first in the join order. The second constraint refers to* $T$ *and* $U$*. The binary representation of the partition ID encodes the concrete set of constraints to use. For the considered worker node, the partition ID is* $10$ *in binary representation. The first bit of the binary representation is zero so the worker node orders* $R$ *before* $S$*. As the second bit is one, the worker orders* $U$ *before* $T$*. Note that other workers will use complementary constraint sets based on their respective partition ID such that the whole join order space is covered. The worker that we focus on finds the best plan whose join order complies with the given constraints. It returns that plan to the master which compares the plans returned by all workers to determine the globally optimal plan.*

We present pseudo-code for the high-level algorithm that is executed by the master and the worker nodes in Section 6.4.1. The code of the sub-functions that the workers use to infer constraints on the join order from the partition ID and to find join orders that comply with the constraints are discussed in Section 6.4.2.

```
// Parallelizes optimization of query $Q$ over $m$ machines.
function MASTER($Q, m$)
    // Generate best plan for each partition in parallel
    parfor $partID \in \{1,\ldots,m\}$ do
        $bestInPart[partID] \leftarrow$WORKER($Q, partID, m$)
    end parfor
    // Prune plans and returns best plan
    $bestPlan \leftarrow bestInPart[1]$
    for $partID \in \{2,\ldots,m\}$ do
        FINALPRUNE($bestPlan, bestInPart[partID]$)
    end for
    return $bestPlan$
end function
```

Algorithm 14 – Function executed by master node for parallel query optimization on shared-nothing architectures.

### 6.4.1 High-Level Algorithm

We present pseudo-code for the high-level algorithms that are executed on the master node and on the workers. As it is common in the area of query optimization, we simplify the presented pseudo-code by considering only SPJ queries and by neglecting for instance the impact of interesting tuple orders [116]. There are however standard methods by which such algorithms can be extended to support richer query languages [116] (e.g., queries with aggregates or nested queries). It is straight-forward to extend the presented algorithm to consider interesting tuple orderings, too.

As announced before, we present two algorithm variants, one treating the space of left-deep plans, the other one treating the space of bushy plans. The pseudo-code that we discuss in this subsection is however the same for both variants such that we do not need to distinguish between them.

Our algorithm consists of two parts: the first part is executed by the master node which orchestrates the worker nodes. The second part of our algorithm runs on the worker nodes. Algorithm 14 shows the code that is executed on the master. The input is a query $Q$, for which we want to find an optimal query plan, and the number $m$ of available worker nodes. We assume in the following that $m$ is a power of two (the reason will become apparent in the following). The output of the MASTER function is the optimal plan for $Q$.

The master node executes two phases. In the first phase, the master sends the query together with a unique partition ID to each of the workers. We discuss the pseudo-code of the WORKER function a bit later. All worker invocations happen in parallel as indicated by the keyword **parfor**. The partition ID identifies a partition of the plan search space. The task of each worker is to find the optimal plan within its respective partition and to return it to the master. The master collects the returned plans in the array $bestInPart$ (we use the standard notation

```
// Generate best plan for query $Q$ in partition with
// ID $partID$ out of $m$ partitions.
function WORKER($Q, partID, m$)
    // Decode partition ID into a set of constraints
    $constr \leftarrow$ PARTCONSTRAINTS($Q, partID, m$)
    // Generate admissible intermediate results
    $joinRes \leftarrow$ ADMJOINRESULTS($Q, constr$)
    // Initialize best plans for single tables
    for $q \in Q$ do
        $P[q] \leftarrow$ SCAN($q$)
    end for
    // Iterate over join result cardinality
    for $k \in \{2, \ldots, |Q|\}$ do
        // Iterate over admissible join results
        for $q \in joinRes : |q| = k$ do
            // Try splits of $q$ into two join operands
            TRYSPLITS($q, constr, P$)
        end for
    end for
    // Return best plan for query $Q$
    return $P[Q]$
end function
```

Algorithm 15 – Generate best query plan within specific partition of either linear or bushy plan space.

$bestInPart[x]$ to represent an access to the $x$-th field of that array). In the second phase, the master node compares all collected plans to identify the globally-optimal plan. Function FINALPRUNE, whose pseudo-code we do not specify, represent a standard pruning function that replaces $bestPlan$ by the better plan among the two input plans. Having considered all plans returned by the workers, the best plan must be globally optimal.

Algorithm 15 shows the code of the function that runs on worker nodes and is invoked by the master. The input is the query $Q$ to optimize, the total number $m$ of plan space partitions, and the identifier $partID$ of the partition that is assigned to the respective worker. The output is the optimal plan within the corresponding partition. Each worker node executes the following three steps. First, knowing the total number $m$ of partitions, the specific partition ID $partID$ can be translated into a set of constraints on the join order. Function PARTCONSTRAINTS, whose code is discussed later, accomplishes the translation. Second, function ADMJOINRESULTS translates the set of constraints into an admissible set of table sets that can appear as join results within a query plan whose join order respects the constraints. Finally, the worker node uses a dynamic programming approach to find the optimal query plan among all plans that produce only admissible join results. We assume, without explicitly writing out the corresponding code, that the result sets generated by function ADMJOINRESULTS have been indexed by their cardinality such that Algorithm 15 can efficiently retrieve all sets with a given cardinality $k$.

Variable $P$ is an array storing optimal query plans and $P[Q]$ designates the optimal query plan for joining the table set $Q$. We initialize $P$ by inserting the scan plan for each single query table $q \in Q$. We simplify the pseudo-code by assuming only one scan plan per table but the generalization is straight-forward. After that, the algorithm calculates optimal plans for table sets of increasing cardinality, using the optimal plans that were stored in prior iterations. The algorithm considers only table sets that represent admissible join results. For each admissible join result, function TRYSPLITS tries all ways of splitting the join result into two admissible operands and stores the best resulting plan in $P$.

### 6.4.2 Plan Space Partitioning

We discuss in the following the sub-functions that are invoked by the WORKER function. In contrast to the previous subsection, we now need to distinguish between the two algorithm variants that we present. In the following pseudo-code, we use the notation F$[LINEAR]$ to indicate that function F is specific to the algorithm searching linear (or left-deep) search spaces. Analogously, F$[BUSHY]$ indicates a function that is specific to the algorithm generating bushy plans. The code of all other functions is the same for both variants.

Algorithm 16 shows the code for translating a partition ID into a set of constraints. Function PARTCONSTRAINTS obtains as input the query, the number of partitions, and the partition ID. The output is a set of constraints on the join order that define the plan space partition that the current worker needs to treat.

When generating constraints, we use the notation $Q_x$ with $x \in \mathbb{N}$ to designate the $x$-th table in query $Q$. This notation assumes that query tables have been numbered consecutively from 0 to $|Q|-1$. The algorithm can use an arbitrary table numbering but it is important that all workers use the same numbering in order to guarantee that the whole plan space is covered by the ensemble of workers.

The form of the generated constraints differs depending on whether we search for left-deep or bushy plans. Constraints for the left-deep plan space are defined on table pairs while constraints on bushy plans are defined on triples of tables. Constraint restricting the linear plan space are of the form $Q_x \prec Q_y$. This means that the $x$-th table must appear before the $y$-th table in an admissible join order (the join order of a left-deep plan can be represented as a sequence of tables and the constraints refer to that representation). Constraints restricting bushy plan spaces are of the form $Q_x \preceq Q_y|Q_z$ with the semantic that when considering the intermediate join results containing table $Q_z$ in ascending order of cardinality, table $Q_y$ must not appear before table $Q_x$. We assume that constraints have been indexed such that all constraints concerning a given set of tables can be retrieved efficiently.

In case of a left-deep plan space there are two complementary constraints for each pair of tables, namely $Q_x \prec Q_y$ and $Q_y \prec Q_x$. In order to guarantee that the whole plan space is

```
// Generate constraint on $i$-th table pair of
// query $Q$ using precedence order $precOrd$.
function CONSTRAINT[LINEAR]($Q, i, precOrd$)
    if $precOrd = 0$ then
        return $Q_{2\cdot i} \prec Q_{2\cdot i+1}$
    else
        return $Q_{2\cdot i+1} \prec Q_{2\cdot i}$
    end if
end function
// Generate constraint on $i$-th table tuple of
// query $Q$ using precedence order $precOrd$.
function CONSTRAINT[BUSHY]($Q, i, precOrd$)
    if $precOrd = 0$ then
        return $Q_{3\cdot i} \preceq Q_{3\cdot i+1}|Q_{3\cdot i+2}$
    else
        return $Q_{3\cdot i+1} \preceq Q_{3\cdot i}|Q_{3\cdot i+2}$
    end if
end function
// Decode partition ID $partID$ into a set of constraints
// restricting the plan space for query $Q$. The total
// number of partitions is $m$ and $partID \leq m$.
function PARTCONSTRAINTS($Q, partID, m$)
    // Initialize constraint set
    $constr \leftarrow \emptyset$
    // Iterate over constraints
    for $i \in \{0, \ldots, \log_2(m) - 1\}$ do
        // $i$-th bit encodes precedence order
        $precOrd \leftarrow$BIT($partID, i$)
        // Generate constraint on $i$-th subset of $Q$
        $c \leftarrow$CONSTRAINT($Q, i, precOrd$)
        // Add new constraint into set
        $constr \leftarrow constr \cup c$
    end for
    return $constr$
end function
```

Algorithm 16 – Translate the partition ID into a set of constraints that restrict the plan search space.

covered by the ensemble of workers, we need to consider complementary constraints by different workers. All workers use constraints on the same table pairs but the direction of those constraints (which of the two tables to join first) differs among workers. Each worker uses the binary representation of the partition ID to derive which of the two possible constraints to consider for each table pair. We use the notation BIT($partID, i$) to represent the $i$-th bit of the binary representation (it does not matter whether we start with the lowest order bits or with the highest order bits). Each bit determines the direction for one constraint.

The treatment of bushy plan spaces is analogue. Constraints are defined on table triples but for each triple of tables there are still just two complementary constraints and each worker picks between them based on the partition ID. We define two variants of the function CONSTRAINT that generates the actual constraints: one for the linear and one for the bushy plan space. The high-level algorithm for generating constraint sets does not differ between them.

Note that we have assumed that the number of workers is a power of two and that the number of query tables is a multiple of two for left-deep plans and a multiple of three for bushy plans. Those assumptions simplify our pseudo-code while the extension to the general case (i.e., using only a subset of workers whose cardinality is a power of two) are straight-forward. The number of workers that can be efficiently exploited by our algorithm is however indeed restricted to powers of two and the maximal number of workers is additionally restricted as a function of the query size. We analyze those restrictions in more detail in Section 6.5.

Constraints restrict the admissible join orders and join trees. We are however ultimately interested in restricting not the number of join orders but rather the number of intermediate results, i.e. join result table sets, that can appear in admissible plans. We focus on reducing the number of result table sets as the time and space complexity of the dynamic programming algorithm executed by the workers depends on it.

We must translate sets of constraints into sets of intermediate results that admissible plans can use. Algorithm 17, more precisely function ADMJOINRESULTS, accomplishes the translation. The input is the query and a set of constraints. The output is the set of intermediate results that can appear in plans that comply with those constraints.

Function ADMJOINRESULTS iterates over all subsets of query tables that constraints can refer to. For left-deep plans those are all pairs of tables with consecutive numbers. For bushy plans those are all triples of consecutive tables. In each iteration of the for loop, the function extends the admissible table sets stored in $R$ by subsets of the table pair (or table triple) considered in the current iteration using a Cartesian product for the extensions. The auxiliary function CONSTRAINEDPOWERSET returns for a given pair (respective triple) or tables all subsets that comply with the constraints. More precisely, if table $Q_x$ needs to be joined before table $Q_y$ in case of left-deep plans then (non-singleton) table sets containing $Q_y$ but not table $Q_x$ do not need to be considered. Equally for bushy plans, if table $Q_x$ must appear before table $Q_y$ when enumerating all table sets containing $Q_z$ then table sets containing $Q_y$ and $Q_z$ but not $Q_x$ are not admissible as join results.

**Example 14.** *Assume that $Q = \{Q_1, Q_2, Q_3, Q_4\}$ and that we have the two constraints $C = \{Q_1 \prec Q_2, Q_4 \prec Q_3\}$, hence we consider left-deep plans. Then the set of admissible join result sets is generated in function* ADMJOINRESULTS *as follows. In the first iteration of the for loop, we extend the elements contained in R (initially this is only the empty set) with the admissible subsets of the first table pair $\{Q_1, Q_2\}$. The admissible subsets are $\{\{\}, \{Q_1\}, \{Q_1, Q_2\}\}$ and*

```
// Returns pairs of consecutive tables in query $Q$
function SUBSETS[LINEAR]($Q$)
    return $\{\{Q_{2\cdot i}, Q_{2\cdot i+1}\} | 0 \leq i \leq |Q|/2-1\}$
end function
// Returns triples of consecutive tables in query $Q$
function SUBSETS[BUSHY]($Q$)
    return $\{\{Q_{3\cdot i}, Q_{3\cdot i+1}, Q_{3\cdot i+2}\} | 0 \leq i \leq |Q|/3-1\}$
end function
// Part of power set of $S$ respecting constraints $C$
function CONSTRAINEDPOWERSET[LINEAR]($S, C$)
    return POWER($S$)$\setminus\{\{Q_y\} | (Q_x \prec Q_y) \in C\}$
end function
// Part of power set of $S$ respecting constraints $C$
function CONSTRAINEDPOWERSET[BUSHY]($S, C$)
    return POWER($S$)$\setminus\{\{Q_y, Q_z\} | (Q_x \preceq Q_y | Q_z) \in C\}$
end function
// Returns all potential join results (table subsets
// of query $Q$) that comply with constraints $C$.
function ADMJOINRESULTS($Q, C$)
    // Initialize result sets
    $R \leftarrow \{\emptyset\}$
    // Iterate over subsets of $Q$
    for $S \in$SUBSETS($Q$) do
        // Extend join results using Cartesian product
        $R \leftarrow R \times$CONSTRAINEDPOWERSET($S, C$)
    end for
    return $R$
end function
```

Algorithm 17 – Generate all table subsets that comply with the constraints defining a search space partition.

*this is at the same time the content of R after the first iteration. The algorithm considers admissible subsets of* $\{Q_3, Q_4\}$ *in the second iteration (which are the sets* $\{\}$, $\{Q_4\}, \{Q_3, Q_4\}$*) and extends each element with all of the admissible subsets. Hence* $R = \{\{\}, \{Q_1\}, \{Q_1, Q_2\}, \{Q_4\}, \{Q_1, Q_4\}, \{Q_1, Q_2, Q_4\}, \{Q_3, Q_4\}, \{Q_1, Q_3, Q_4\}, \{Q_1, Q_2, Q_3, Q_4\}\}$ *after the second iteration.*

Note that the admissible table sets generated by function ADMJOINRESULTS do not include all singleton table sets. While all singleton sets must be considered to generate any plan (since we need to select scan plans for each table), singleton sets are treated separately in Algorithm 15 and it does not matter which of them are included in the result of function ADMJOINRESULTS.

Algorithm 18 shows the function trying out different splits and generating corresponding plans that applies for left-deep plans. This function is called by Algorithm 15 for each admissible

```
// Try all splits of $U \subseteq Q$ into two operands respecting
// constraints $C$, generate associated plans and prune.
function TrySplits[Linear]$(Q, U, C, P)$
    // Iterate over potential inner operands
    for $u \in U$ do
        // Check if operand choice satisfies constraints
        if $\nexists v \in U : (u \prec v) \in C$ then
            $p \leftarrow$Join$(P[U \setminus u], P[u])$
            Prune$(P, p)$
        end if
    end for
end function
// Try all splits of $U \subseteq Q$ into two operands respecting
// constraints $C$, generate associated plans and prune.
function TrySplits[Bushy]$(Q, U, C, P)$
    // Determine admissible operands
    $A \leftarrow \{\emptyset\}$
    // Iterate over set of table triples
    for $T \in$Subsets[Bushy]$(Q)$ do
        // Restrict triple to tables in join result
        $S \leftarrow T \cap U$
        // Form power set of remaining triples
        $S \leftarrow$Power$(S)$
        // Take out sets violating constraints
        $S \leftarrow S \setminus \{\{Q_y, Q_z\} | (Q_x \preceq Q_y | Q_z) \in C\}$
        // Remove complement of inadmissible sets
        $S \leftarrow S \setminus \{\{Q_x\} | (Q_x \preceq Q_y | Q_z) \in C; Q_y, Q_z \in U\}$
        // Extend admissible splits by Cartesian product
        $A \leftarrow A \times S$
    end for
    // Full set and empty set do not qualify as operands
    $A \leftarrow A \setminus \{\emptyset, U\}$
    // Iterate over admissible left operands
    for $L \in A$ do
        // Generate plans associated with splits
        $p \leftarrow$Join$(L, U \setminus L)$
        // Discard suboptimal plans
        Prune$(P, p)$
    end for
end function
```

Algorithm 18 – Generate and prune query plans that correspond to different splits of a join result into two operands.

join result. The function iterates over all tables in the join result set $U$ and tries all of them as inner join operands as long as none of the constraints is violated. Plans corresponding to admissible splits are generated and function PRUNE, whose pseudo-code we do not specify, compares the newly generated plan against the best plan known so far that produces the same intermediate result as the new one. Only the better one of those two plans remains in the result set. Note that the pruning function can store one optimal plan for each interesting tuple ordering [116]. The pruning function used by the workers might differ from the one used by the master (called FINALPRUNE in Algorithm 14) as the tuple ordering is for instance only relevant as long as it can reduce the cost of future operations and does not need to be taken into account anymore for completed plans.

There are actually two mechanisms by which partitioning reduces the time complexity per worker. So far we have focused on the first one: partitioning reduces the time complexity per worker since fewer potential join results need to be considered. An additional advantage of partitioning is however that it allows to reduce the number of splits of join results into two join operands, leading to different query plans that need to be generated and compared.

The potential for saving computation time by reducing the number of splits is higher for bushy plan spaces since the number of possible splits grows exponentially in the size of the join result. For left-deep plans, the number of splits grows only linearly in the cardinality of the join result as the right join operand is limited to singleton table sets.

This is why we invest more effort in case of bushy than in case of left-deep plans into properly exploiting the reduction of admissible splits. For left-deep plans, we basically enumerate all possible splits and check whether they comply with the constraints. The complexity of that approach remains linear in the number of possible splits and not in the lower number of admissible splits. The algorithm for bushy plans is more sophisticated as it avoids generating non-admissible splits for bushy plans in the first place. Hence its complexity is linear in the number of admissible rather than possible splits.

Function TRYSPLITS[BUSHY] generates all admissible splits in a bushy plan space and generates and prunes the associated query plans. The algorithm first generates all admissible join operands and stores them in variable $A$. Each admissible join operand corresponds to the union of one admissible subset for each table triple (constraints are defined on triples of tables). This is why we iterate over all table triples, determine all admissible subsets of the current triple, and combine in each iteration each operand in $A$ with each admissible subset of the current triple (using a similar approach as in Algorithm 17). For a given triple of query tables, we only consider the ones that are included in the join result $U$ that needs to be split. If no constraints are defined on the current triple then the entire power set of the contained table is admissible. Otherwise, we must remove subsets violating the precedence constraints (line 25) but we must also remove subsets whose complement (in the contained triple tables) would violate the precedence constraints (line 27) as the second join operand is the complement of the first operand.

Having determined all admissible join operands (whose complement is admissible, too), we iterate over all of them, generate plans and discard sub-optimal plans.

## 6.5 Complexity Analysis

We analyze the asymptotic complexity of the algorithm presented in the previous section according to multiple metrics: we analyze the asymptotic amount of data sent over the network in Section 6.5.1, the consumed amount of main memory in Section 6.5.2, and the execution time in Section 6.5.3.

We simplify the following analysis by assuming that only one scan and join operator is used. We also assume that only one cost metric is considered when comparing query plans and that no interesting orders are present. In Section 6.5.4, we discuss how the analysis can be extended. In Section 6.5.5 we discuss the question of whether our partitioning methods can be improved and show that they are optimal at least within a restricted space of partitioning methods.

We introduce notations that are used throughout this section. We denote by $n = |Q|$ the number of query tables to join and by $m$ the number of worker machines. We assume that $m \leq 2^{\lfloor n/2 \rfloor}$ for linear plan search spaces and $m \leq 2^{\lfloor n/3 \rfloor}$ for bushy plan spaces. This is required as we assume that the table sets that different constraints refer to are mutually disjunct. We use two tables per constraint for linear plan spaces and three tables for bushy plan spaces. We denote by $l = \lfloor \log_2(m) \rfloor$ the number of constraints per plan space partition. By $b_q$ we designate the byte size of the input query. By $b_p$ we denote the byte size of a corresponding query plan.

### 6.5.1 Network Communication

We analyze the asymptotic size of the data that is sent over the network during optimization of one query.

**Theorem 26.** *The amount of data sent over the network is in $O(m \cdot (b_q + b_p))$.*

*Proof.* Different workers do not communicate with each other so data is only sent between master and workers. This happens at two occasions: when assigning each worker to a plan space partition and when collecting the best plans for each partition. The input for each worker is the query (with space consumption $b_q$) and two integer numbers with constant space consumption. We consider one plan cost metric and no interesting orders (while the extensions are discussed later). The output of each worker is therefore a single query plan with space consumption $b_q$. □

### 6.5.2 Main Memory

We analyze the amount of main memory that each worker requires during optimization. Note that the main memory consumption of the master is negligible as it delegates optimization. The main memory consumed per worker node depends on the number of admissible join results.

**Theorem 27.** *Each linear plan space partition restricted by $l$ constraints has $O(2^n \cdot (3/4)^l)$ admissible join results.*

*Proof.* The proof is an induction over the number of constraints $l$. For $l = 0$ (induction start), all subsets of $Q$ are admissible and their number is in $O(2^n)$. Assume the induction holds up to $L$ constraints. We will see that it holds for $L+1$ constraints as well. All constraints refer to different tables. Hence the first $L$ constraints do not influence the occurrence frequency of the two tables $x$ and $y$ that the $L+1$-th constraint refers to. More precisely, among the table sets that remain admissible after considering the first $L$ constraints, the fraction of table sets containing $x$ and $y$, $x$ but not $y$, $y$ but not $x$, and neither $x$ nor $y$, is always $1/4$. Denote by $x \prec y$ the $L+1$-th constraint stating that we must join $x$ before $y$. Then join results containing $y$ but not $x$ are inadmissible, the number of admissible table sets is reduced by factor $3/4$, and the induction holds. □

**Theorem 28.** *Each bushy plan space partition restricted by $l$ constraints has $O(2^n \cdot (7/8)^l)$ admissible join results.*

*Sketch.* The proof follows closely the one of Theorem 27 with the difference that each constraint of the form $x \preceq y|z$ excludes all table sets that contain $y$ and $z$ but not $x$ and their fraction is always $1/8$ among the table sets satisfying all other constraints. □

We use the number of admissible join results to calculate main memory consumption.

**Theorem 29.** *The main memory consumption per node is in $O(2^n \cdot (3/4)^l)$ for linear plan spaces and $O(2^n \cdot (7/8)^l)$ for bushy plan spaces.*

*Proof.* The main memory consumption per worker dominates the consumption of the master. The variable with dominant space consumption are the ones storing admissible join results (variable $joinRes$ in Algorithm 15) and the one assigning table sets to optimal plans (variable $P$). We currently assume one plan cost metric and therefore only one plan is optimal per table set. Storing plans generally takes $O(n)$ space but here each plan can be represented by at most two pointers to optimal sub-plans stored for table subsets which requires only $O(1)$ space. The total main memory consumption follows from Theorems 27 and 28. □

### 6.5.3 Execution Time

We analyze time complexity. Note that the pseudo-code presented in Section 6.4 is rather abstract and does not contain certain steps that are crucial for efficiency: as we mentioned in Section 6.4, we assume for instance that constraints are indexed such that we can find all constraints in which a given table appears in constant time. For the following analysis, we assume that such commonsense optimizations have been applied (we use them as well in our implementation that is evaluated in Section 6.6).

We first analyze execution time on the master.

**Theorem 30.** *The master requires $O(m \cdot (b_q + b_p))$ time.*

*Proof.* The master distributes the query and the partition ID to all $m$ workers. Assuming that the required time is proportional to the amount of data being sent, distributing tasks takes $O(mb_q)$ time and collecting plans from the workers is in $O(mb_p)$. After receiving all plans, the master iterates over the $m$ plans that were returned from the workers (and whose cost was already calculated) and determines the one with minimal cost. This has complexity $O(m)$. □

Now we analyze the time complexity of processing a linear plan space partition.

**Theorem 31.** *The time complexity for processing a linear plan space partition by one of the workers is $O(n \cdot 2^n \cdot (3/4)^l)$.*

*Proof.* A worker performs three main steps per invocation: translating the partition ID into constraints, translating constraints into admissible join result sets, and determining the optimal plan among the plans using only admissible join results. The operation with dominant time complexity is the determination of the optimal plan. For each admissible join result set, we iterate over less than $n$ inner join operands. The number of admissible join result sets is in $O(2^n \cdot (3/4)^l)$ according to Theorem 27. Generating a plan from two sub-plans, calculating its cost via recursive formulas, and comparing it with the best previously generated plan joining the same tables requires only constant time. □

**Theorem 32.** *The time complexity for processing a bushy plan space partition by one of the workers is $O(3^n \cdot (21/27)^l)$.*

*Proof.* Determining the optimal plan in the restricted plan space partition is the operation with dominant time complexity. The time complexity for finding an optimal plan is lower-bounded by the number of considered result table sets. It is proportional to the number of considered join operand pairs.

For each table there are in general three possibilities for how it appears in a pair of join operands: either it appears in the left operand or in the right operand or it does not appear (neither in the operands nor in the join result). Join operands are constructed from admissible subsets of table triples. If no constraint is defined on a given triple then all splits are admissible which makes $3^3 = 27$ possible pairs. If a constraint is defined on a triple then some of those 27 possibilities are not admissible. If the constraint is $x \preceq y|z$ then the following six splits of triple $\{x, y, z\}$ are excluded: all splits whose union contains $y$ and $z$ but not $x$ (this applies to four splits) and all splits that assign $y$ and $z$ to one operand and $x$ to the other one (this applies to two splits). The ratio of admissible to possible splits is therefore 21/27 for each triple with a constraint on it. □

As the time complexity of the worker processes dominates the complexity of the master process and as all workers execute in parallel, the time complexity of a single worker is the complexity of the entire optimization process.

### 6.5.4 Extensions

So far we considered one plan cost metric, no interesting orders, and no alternative operator implementations. It is however straight-forward to extend the analysis as we sketch out in the following.

Considering multiple alternative operator implementations for scan and join operations influences only time complexity. Time complexity grows linearly in the number of operators as each join operator implementation must be considered for each possible pair of join operands. Annotating the operations within query plans by an operator ID does neither change the asymptotic space complexity in main memory nor the asymptotic communication overhead as storing an integer ID requires constant space.

Considering interesting orders or considering multiple plan cost metrics both have the effect that multiple plans can be optimal for joining the same set of tables. The number of interesting orders restricts the number of plans that need to be stored per table set. Assuming that multiple plan cost metrics are considered while using an approximation factor, the number of Pareto-optimal plans per table set can be bounded as shown in Chapter 2. The number of plans sent from workers to master, and therefore the communication overhead, increases linearly in the number of plans stored per table set. Main memory consumption also increases linearly in the number of plans. Time complexity increases proportional to the cube of the number of plans for the following reason: when searching for the optimal plan within each plan space partition, we need to consider all pairs of optimal plans for each split of a table set into two join operands. This accounts for a quadratic increase in complexity. Additionally, pruning takes longer as we need to compare plans not against one but multiple optimal plans. Together this implies a cubic increase in complexity. Note that plans need to be compared according to multiple cost metrics but the number of plan cost metrics is usually considered a constant, as justified in Chapter 2.

### 6.5.5 Optimality of Partitioning

Execution time and main memory consumption both depend on the number of intermediate join results that need to be treated by each worker. With our partitioning scheme, the number of join results per worker reduces by factor 3/4 in case of linear plan spaces and by factor 7/8 for bushy plans, each time that the number of workers doubles. As the ideal factor of 1/2 is not reached there must be join results that are assigned to multiple workers. This raises the question of whether we can do better and reduce the number of intermediate results per worker node by a lower factor.

We answer that question in the following for partitioning methods that are similar to the one we apply. Similar methods are methods that divide the power set of query tables into subsets based on which out of two, respective three, fixed tables are present. Each of the resulting subsets is assigned to part of the workers and each worker generates all plans whose join results are contained in its assigned subsets (each worker constructs scan plans for all single tables, independently from the assigned join results). Workers do not exchange partial plans and hence must generate completed plans and start optimization from scratch.

The following theorems study the best speedup that is achievable by partitioning the plan space between two workers. The reasoning can however be generalized to higher degrees of parallelism.

**Theorem 33.** *Doubling the number of workers cannot reduce the maximal number of join results per worker by less than factor 3/4 in linear plan spaces.*

*Proof.* For a fixed pair of tables $\{x, y\}$ out of all query tables, we denote by $\overline{x}y$ the set of table sets containing $y$ but not $x$, by $xy$ the sets containing both tables, by $\overline{xy}$ sets containing neither $x$ nor $y$, and by $x\overline{y}$ the remaining sets. Each worker must obtain subset $xy$ in order to generate complete plans. The cardinality of the set of joined tables can only increase by one from one join to the next in a left-deep plan space. Each worker needs therefore either join results from $\overline{x}y$ or from $x\overline{y}$ in order to generate any valid plan. Set $\overline{xy}$ must be assigned to at least one worker since the plan space partitioning is otherwise incomplete. □

**Theorem 34.** *Doubling the number of workers cannot reduce the maximal number of join results per worker by less than factor 7/8 in bushy plan spaces.*

*Sketch.* For a triple of tables $\{x, y, z\}$, we use a similar notation as before to characterize join result sets and denote for instance by $x\overline{y}z$ all sets containing $x$ and $z$ but not $y$. Both workers require $xyz$ for the same reason as before. Assume that we do not assign the set $\overline{xyz}$ to both workers. The worker to which $\overline{xyz}$ is assigned is the only worker that can consider plans joining the other tables besides $x$, $y$, and $z$ independently before joining with the triple tables. This means that this worker needs to cover all possible join orders for $x$, $y$, and $z$. Hence it requires all join result sets which defeats the purpose of partitioning.

Assume now that we do not assign the set $x\overline{yz}$ to the first worker. Then the second worker is the only one that can consider plans of the form $(x \bowtie \ldots) \bowtie (y \bowtie \ldots)$ and hence requires $\overline{x}y\overline{z}$ and by analogue reasoning also $\overline{xy}z$ in addition to $x\overline{yz}$ in order to make sure that the whole plan space is covered. As the second worker is at the same time the only one that can consider plans of the form $((x \bowtie \ldots) \bowtie y) \bowtie \ldots$, it requires at the same time $xy\overline{z}$ and $x\overline{y}z$. Since only the second worker can treat plans of the form $(x \bowtie \ldots) \bowtie (y \bowtie z)$, it requires also

$\overline{x}yz$. So the second worker obtains at least 7 sets of join results. The same happens when not assigning $\overline{x}y\overline{z}$ or $\overline{x}\overline{y}z$ to the first worker. We have the option of not assigning one of the three sets containing two out of the three tables $\{x, y, z\}$ to the first worker in which case we need to assign the other two to the second worker. The maximal number of intermediate result splits per worker remains $7/8$. □

## 6.6 Experimental Evaluation

We evaluate the scalability of our query optimization algorithm on a large cluster with 100 nodes and parallelize over up to 256 Spark executors. Parallelizing query optimization on clusters is useful if query plans are also executed on a cluster: it is preferable to use all available resources for optimization instead of leaving nodes idle until serial optimization finishes. For queries that are executed regularly, a cluster can be used before run time to calculate optimal query plans if the search space is too large for optimization on a single node. While parallelizing query optimization on a cluster is hence a relevant application scenario, we also selected it specifically because it is a very challenging scenario for parallelization due to high communication cost and setup overhead. The fact that our algorithm scales even on a cluster provides in our opinion strong evidence for that it scales in a multitude of other scenarios as well. Our algorithm is not restricted to shared-nothing architectures but can also be applied in shared-memory settings.

We evaluate our algorithm, in comparison with a baseline, for traditional query optimization with one plan cost metric as well as for multi-objective query optimization where query plans are compared according to multiple cost metrics. We also calculate the speedups that we obtain by parallelization compared to serial algorithms [116, 137]. Section 6.6.1 describes our experimental setup and Section 6.6.2 our experimental results.

### 6.6.1 Experimental Setup

We evaluate our algorithm on a cluster with 100 nodes. Each node is equipped with two Intel Xeon E5-2630 v2 CPUs featuring six cores each running at 2.60GHz; 128 GB of main memory and 20 TB of hard disk capacity are available per node. The cluster runs Ubuntu Linux, version 14.04.

All benchmarked algorithms use Spark 1.5 on Yarn 2.7.1 and are implemented in Java 1.7. We implemented the algorithm from Section 6.4 and abbreviate it by MPQ (for massively parallel query optimization) in the following plots. We compare against an algorithm that is representative for the rather fine-grained approaches to parallelizing query optimization proposed so far. They were targeted at shared-memory architectures and moderate degrees of parallelism [67, 68]. We call that algorithm SMA (for shared-memory approach) in the following plots. In this algorithm, the master node assigns to each worker a set of join results for which to calculate optimal plans based on the optimal plans that were generated by other workers. This means that intermediate results need to be shared between workers and that the master needs to assign multiple rounds of tasks to the workers. The comparison between MPQ and SMA is of course unfair as both were developed for different architectures and different degrees of parallelism. We are however not aware of any other query optimization algorithms targeted at shared-nothing architectures.

We use up to 256 Spark executors and reserve up to 40 GB of main memory per executor (query optimization requires large amounts of memory, in particular in case of multiple plan cost metrics, as shown in Chapter 2). We set the maximum message size to 1 GB (MSA needs to send large messages).

We compare algorithms in linear and bushy plan spaces. We always consider the full plan space and do not heuristically restrict the use of cross products as this might miss optimal plans [106]. As we allow cross products, the number of intermediate results to consider and hence performance of our optimization algorithms does not critically depend on the structure of the query join graphs. We generate random star join graphs, table cardinalities, and predicate selectivity values by the method introduced by Steinbrunn et al. [131] which is commonly used for query optimization benchmarks [31]. In a first series of experiments, we consider execution time as only cost metric and use standard cost formulas [131] to estimate the cost of standard join operators such as block-nested loop join, hash join, and sort-merge join. In a second series of experiments, we consider two plan cost metrics and the goal is hence to approximate the set of Pareto-optimal plans (a plan is Pareto-optimal if no other plan has better cost according to all cost metrics [137]). Our second cost metric (in addition to execution time) is the buffer space consumption such that we calculate optimal cost tradeoffs between execution time and buffer space consumption. We already used those cost metrics in benchmarks in the previous chapters.

For the series of experiments with two plan cost metrics, we replace the standard pruning function in our algorithms by the pruning function introduced in Chapter 2. That pruning function is parameterized by an approximation factor $\alpha$ and we set $\alpha = 10$.

## 6.6.2 Experimental Results

Due to space restrictions, we show only an extract of our full experimental results. The presented results are however representative and we observed the same tendencies in our additional experiments.

We start by discussing the results for traditional query optimization with one plan cost metric. Figure 6.1 shows a comparison between MPQ and MSA in terms of optimization time and in terms of the amount of data exchanged between cluster nodes. Each data point in the plots corresponds to the median of the results for twenty randomly generated queries. We compare algorithms for different plan spaces (linear and bushy) and for different query sizes (number of joined tables). We try different degrees of parallelism for each plan space, adapting

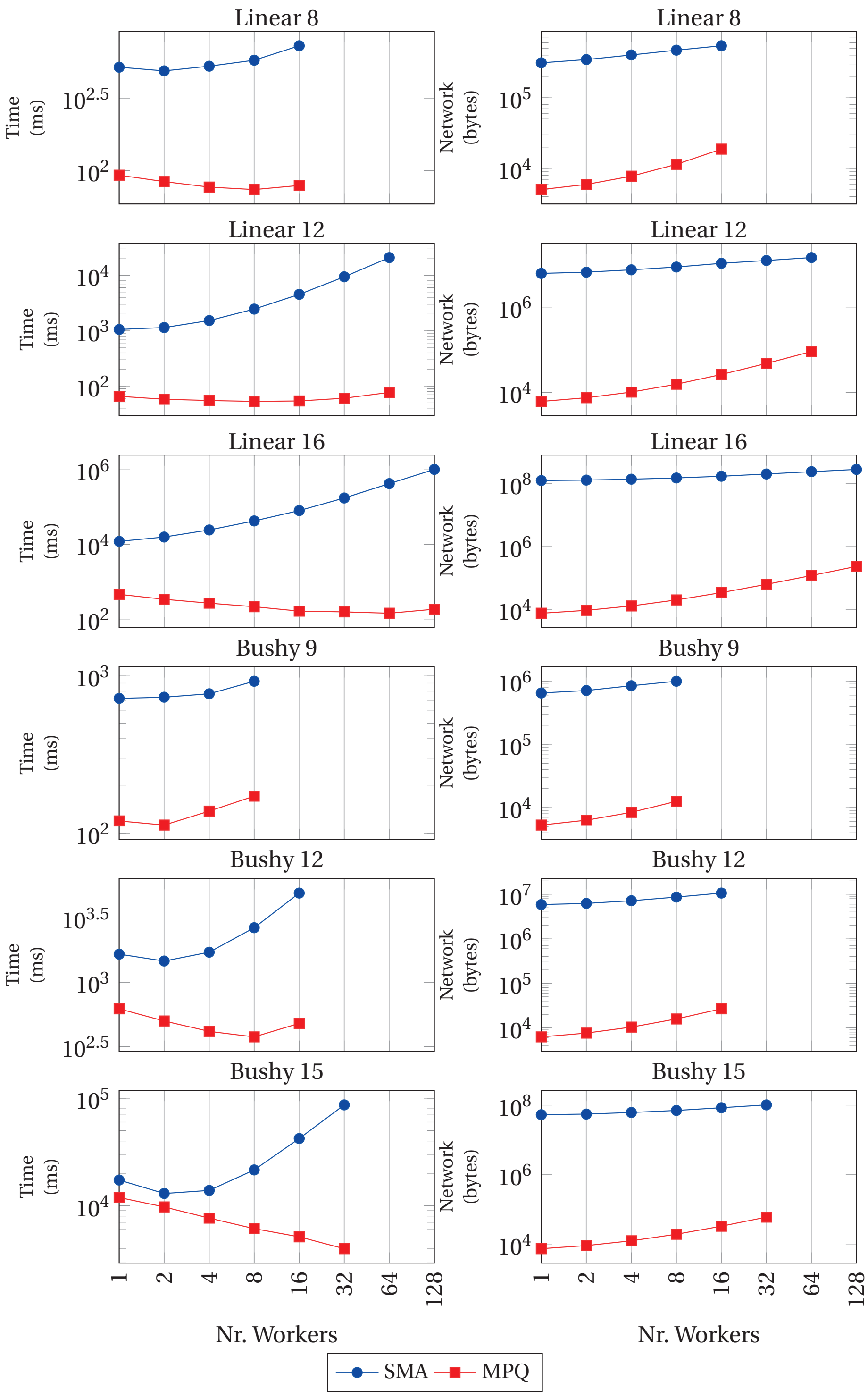


Figure 6.1 – MPQ outperforms MSA by up to four orders of magnitude in terms of optimization time; scalability of MPQ is limited due to the query sizes.

the maximal parallelism to the search space size (we scaled up to the maximal degree of parallelism that MPQ can exploit based on the number of disjoint table pairs or triples) up to a maximum of 128 workers. We try smaller query sizes for the bushy plan space than for the linear plan space as the size of the bushy search space grows faster in the number of query tables. Note that we also consider Cartesian product joins in contrast to prior evaluations of parallel query optimization algorithms. This makes the plan space much larger for the same number of tables. Still the search spaces treated in Figure 6.1 are of moderate size and we try larger search spaces in the following.

MPQ outperforms MSA by up to four orders of magnitude in optimization time. The reason is the large amount of data that MSA has to send over the network, due to the need for sharing intermediate results between workers, and the overheads on the master node by fine-grained task management. The amount of data sent by MSA reaches several hundreds of megabytes while our algorithm sends at most 234 kilobytes and in most cases significantly less than that. As outlined before, MSA is not designed for shared-nothing scenarios and the performance gap between the algorithms is to be expected.

The search space sizes in Figure 6.1 represent approximately the limit of what the competitor algorithm MSA can treat within reasonable amounts of time. For our MPQ algorithm, the considered search spaces are actually too small to justify parallelization. This is why we see in most plots in Figure 6.1 no decrease in optimization time for MPQ with growing degree of parallelism (except for the bushy search space with 15 query tables). The absolute optimization times are for MPQ already very low even for a single worker so parallelization is not needed yet. The amount of network traffic and the management overhead increase for both algorithms once the number of workers increases. MSA can only benefit in few cases from parallelization and only up to a degree of parallelism of four.

The computation time of MSA increases quickly in the query size and in the degree of parallelism as well (reaching more than 15 minutes per test case for 16-table joins). This is why we exclude it from the following series of experiments.

Figure 6.2 shows results for larger search spaces and only for MPQ. The figure shows total optimization time (measured on the master node) as well as the maximal optimization time measured over all workers ("W-Time" in the figure). The fact that the difference between both is small indicates that the management overhead on the master node is negligible. We show network traffic and additionally the maximal main memory consumption over all of the workers (the master does not perform optimization itself and its main memory consumption is negligible). We scale for each query size up to the maximal degree of parallelism supported by our algorithm (determined by the number of table pairs for linear plans and the number of table triples for bushy plans) and maximally up to 128 workers.

As search space sizes are large enough, we see steady scaling for all degrees of parallelism that are theoretically supported by our algorithm without diminishing returns for higher number of workers. The scaling is slightly better for linear plans than for bushy plans which

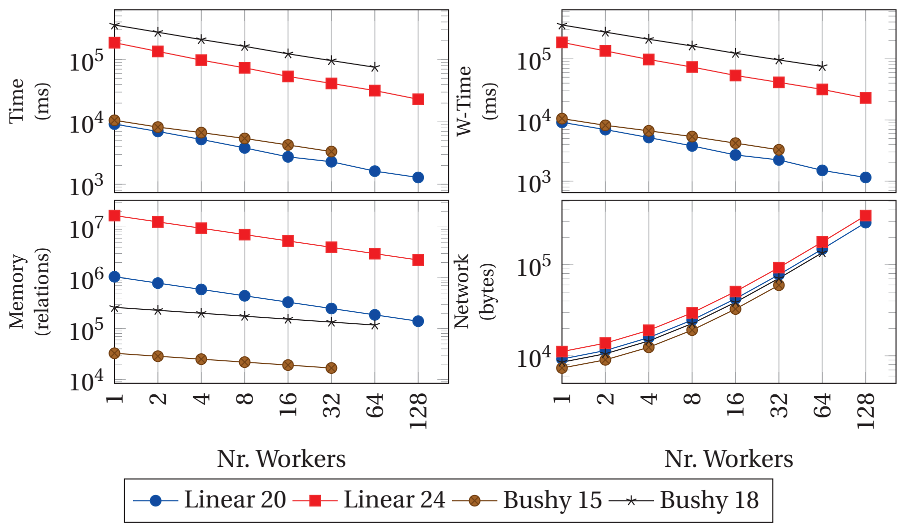


Figure 6.2 – MPQ scales steadily for sufficiently large search spaces and one plan cost metric.

matches precisely our theoretical predictions from Section 6.5 (execution time decreases by factor 3/4 for linear plans but only by factor 21/27 for bushy plans, each time that the degree of parallelism doubles). Unlike for MSA, the network traffic created by MPQ depends only marginally on the query size as no intermediate results have to be exchanged between workers or between workers and master. Only the query itself and the final plan generated by each worker are sent. The maximal main memory consumption on the workers (measured by the number of relations for which to store optimal plans) equally decreases steadily with increasing parallelization. Here the decrease for bushy plans is slower than for linear plans which again matches our theoretical results.

If we use one worker then no constraints on the join order are defined. Then MPQ is equivalent to the classical Selinger algorithm [116] as it treats the same table sets in the same order. Hence we compare the optimization time when executing our algorithm on a single worker (not measuring master computation time and communication overheads) to the optimization time of the parallel version (including master computation time and communication overheads) to obtain the speedup of our algorithm compared to serial query optimization. With 128 workers, we obtain for left-deep plans a speedup of 8.1 for 24 query tables and a speedup of 7.2 for 20 tables. With 32 workers we have a speedup of 3.2 for 15-table joins and bushy query plans and a speedup of 4.8 for 18-table joins and 64 workers.

We finally want to point out that our Java-based implementation is not optimized for maximum efficiency. It is rather optimized for modularity, allowing to "plug-in" different search spaces and cost metrics. This enables us to execute experiments over a broad range of scenarios but it also introduces overheads in some of the functions that are most frequently called during optimization. We believe that optimization efficiency can be significantly improved by specializing the algorithm to a single scenario.

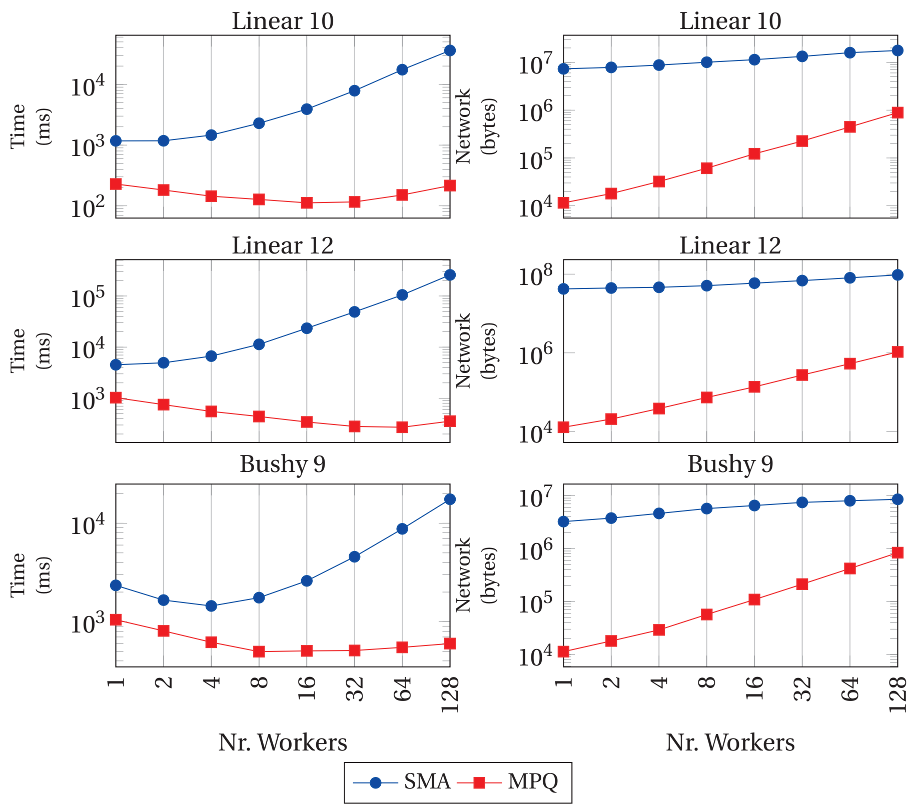


Figure 6.3 – MPQ outperforms MSA but its scalability is limited by small query sizes.

We discuss the results for multi-objective query optimization. Figure 6.3 shows a comparison between multi-objective versions of MSA and MPQ (both algorithms use the same pruning function that we reconfigured to consider two cost metrics). The tendencies are similar as for single-objective query optimization. Optimization times and network traffic are significantly lower for MPQ than for MSA. The network traffic of MPQ has however increased when comparing to the results for single-objective query optimization. The reason is that each worker must now send the set of all Pareto-optimal plans in its respective plan space partition back to the master instead of only one plan. The median number of complete Pareto-optimal plans per query was 21 for 12-table joins when considering left-deep plans and 16 for 9-table joins in a bushy plan space.

Instead of exploiting a high degree of parallelism, MSA suffers significantly once the number of workers increases due to network traffic and coordination overhead. The maximal degree of parallelism that was beneficial to MSA is eight. This is also the number of threads that prior algorithms were maximally evaluated on. MPQ benefits from parallelism up to 32 workers for 10-table joins and left-deep plans, for up to 64 workers for 12-table joins, and for up to eight workers for 9-table joins and bushy plan spaces which corresponds to the number of disjoint table pairs respective triples. The absolute run times of MPQ are however so low that parallelization is unnecessary.

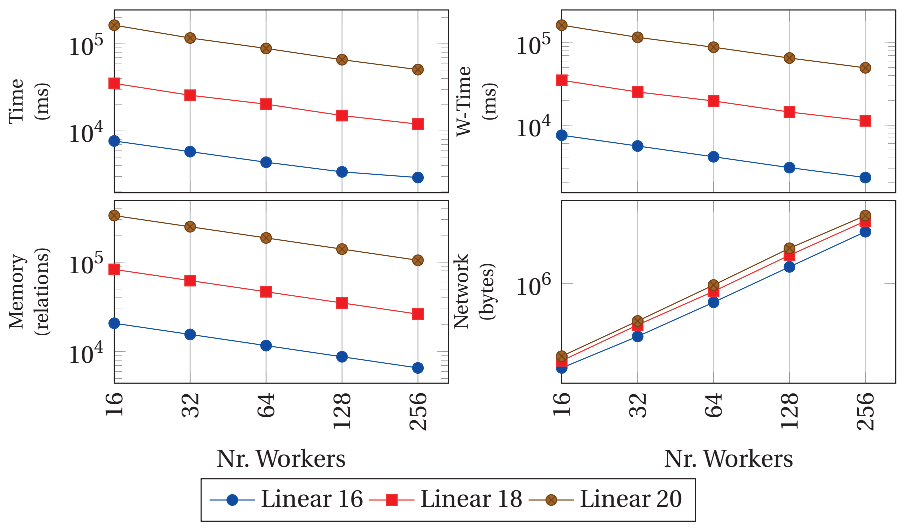


Figure 6.4 – MPQ scales steadily using up to 256 workers for linear plan spaces and two plan cost metrics.

Figure 6.4 shows results for MPQ on queries that are sufficiently large to exploit large degrees of parallelism. The scaling is steady and without noticeable diminishing returns effects up to the maximal number of 256 workers. Note that the run times of MPQ in Figure 6.4 are lower than the run times of MSA in Figure 6.3, even though we consider significantly larger search spaces in Figure 6.4. We tested scalability for bushy plans and more than 9 query tables and saw steady scaling up to the number of table triples in the query. We omit those results due to space restrictions.

Our algorithm is for one worker equivalent to the algorithm from Chapter 2. We calculate speedups in a similar way as before and obtain a speedup of 5.1 for 16-table joins, 5.5 for 18-table joins, and 9.4 for 20-table joins.

## 6.7 Conclusion

We presented a generic plan space decomposition method for query optimization that is applicable for single- and multi-objective query optimization and for other variants. We demonstrated scalability using up to 256 workers.

# 7 Linearization

All methods presented in the prior chapters are similar in that they address the query optimization problem in its original form. In this chapter, we will transform query optimization into a different problem, the problem of mixed integer linear programming (MILP). This has the advantage that we can apply existing software solvers to the transformed problem. We will see that doing so allows treating significantly larger search spaces in query optimization than with dynamic programming based approaches. Furthermore, the experimental results that we see in this chapter represent only snapshots capturing the current state of the art in MILP solver technology. MILP solvers have steadily improved their performance over the past decades (hardware independently). By connecting query optimization to MILP, we will automatically benefit from all future advances in the highly fruitful research area of MILP.

## 7.1 Introduction

From the developer's perspective, there are two ways of solving a hard optimization problem on a computer: either we write optimization code from scratch that is customized for the problem at hand or we transform the problem into a popular problem formalism and use existing solver implementations. In principle, the first approach could lead to more efficient code as it allows to exploit specific problem properties. Also, we do not require a transformation that might blow up the size of the problem representation. In practice however, our customized code competes against mature solver implementations for popular problem models that have been fine-tuned over decades [27], driven by a multitude of application scenarios. Using an existing solver reduces the amount of code that needs to be written and we might obtain desirable features such as parallel optimization or anytime behavior (i.e., obtaining solutions of increasing quality as optimization progresses) automatically from the solver implementation. It is therefore in general advised to consider and to evaluate both approaches for solving an optimization problem.

We apply this generic insight to the problem of database query optimization. For the last thirty years, the problem of exhaustive query optimization, more precisely the core problem of join ordering and operator selection [116], has typically been solved by customized code inside the query optimizer. Query optimizers consist of millions of code lines [145] and are the result of thousands of man years worth of work [82]. The question arises whether this development effort is actually necessary or whether we can transform query optimization into another popular problem formalisms and use existing solvers. We study that question in this chapter.

We transform the join ordering problem into a mixed integer linear program (MILP). We select that formalism for its popularity. Integer programming approaches are currently the method of choice to solve thousands of optimization problems from a wide range of areas [94]. Corresponding software solvers have sometimes evolved over decades and reached a high level of maturity [27]. Commercial solvers such as Cplex[1] or Gurobi[2] are available for MILP as well as open source alternatives such as SCIP[3].

Those solvers offer several features that are useful for query optimization. First of all, they possess the anytime property: they produce solutions of increasing quality as optimization progresses and are able to provide bounds for how far the current solution is from the optimum. Chaudhuri recently mentioned the development of anytime algorithms as one of the relevant research challenges in query optimization [34]. Mapping query optimization to MILP immediately yields an algorithm with that property (note that the anytime algorithm from Chapter 3 is not applicable to traditional query optimization). Second, MILP solvers already offer support for parallel optimization which is an active topic of research in query optimization as well [67, 145, 129]. Finally, the performance of MILP solvers has improved (hardware-independently) by more than factor 450,000 over the past twenty years [27]. It seems entirely likely that those advances can speed up query optimization as well (and anticipating our experimental results, we find indeed classes of query optimization problems where a MILP based approach treats query sizes that are illusory for prior exhaustive query optimization algorithms).

In summary, by connecting query optimization to integer programming, we benefit from over sixty years of theoretical research and decades of implementation efforts. Even better, having a mapping from query optimization to MILP does not only enable us to benefit from past research but also from all future research and development advances that originate in the fruitful area of MILP. Performance improvements have been steady in the past [27] and, as several major software vendors compete in that market, are likely in the future as well.

Given that integer programming transformations have been proposed for many optimization problems that connect to query optimization [13, 20, 54, 108, 148], it is actually surprising that no such mapping has been proposed for the join ordering problem itself so far. There are even sub-domains of query optimization, notably parametric query optimization [59, 73, 74] and multi-objective parametric query optimization (this problem was introduced in Chapter 4), where it is common to approximate the cost of query plans via piecewise-linear functions. The

[1]http://www.ibm.com/software/products/en/ibmilogcpleoptistud
[2]http://www.gurobi.com/
[3]http://scip.zib.de/

purpose here is however to model the dependency of plan cost on unknown parameters while traditional approaches such as dynamic programming are used to find the optimal join order. None of the aforementioned publications transforms the join ordering problem into a MILP and the same applies for additional related work that we discuss in Section 7.2.

A MILP is specified by a set of variables with either continuous or integer value domain, a set of linear constraints on those variables, and a linear objective function that needs to be minimized. An optimal solution to a MILP is an assignment from variables to values that minimizes the objective function. We sketch out next how we transform the join ordering problem into a MILP.

Left-deep query plans can be represented as follows (we simplify by not considering alternative operator implementations while the extensions are discussed later). For a given query, we can derive the total number of required join operations from the number of query tables. As we know the number of required joins in advance, we introduce for each join operand and for each query table a binary variable indicating whether the table is part of that join operand. We add linear constraints enforcing for instance that single tables are selected for the inner join operands (a particularity of left-deep query plans), that the outer join operands are the result of the prior join (except for the first join), or that join operands have no overlap. The result is a MILP where each solution represents a valid left-deep query plan.

This is not yet useful: we must associate query plans with cost in order to obtain the optimal plan from the MILP solver. The cost of a query plan depends on the cardinality (or byte size) of intermediate results. The cardinality of an intermediate result depends on the selected tables and on the evaluated predicates. We introduce a binary variable for each predicate and each intermediate result, indicating whether the predicate has been evaluated to reduce cardinality. Predicate variables are restricted by linear constraints that make it impossible to evaluate a predicate as long as not all query tables it refers to are present in the corresponding result. The cardinality of the join of a set of tables on which predicates have been evaluated is usually estimated by the product of table cardinalities and predicate selectivities. As we cannot directly represent a product via linear constraints, we focus on the logarithm of the cardinality: the logarithm of a product is the sum of the logarithms of the factors. Based on our binary variables representing selected tables and evaluated predicates, we calculate the logarithm of the cardinality for all intermediate results that appear in a query plan. Based on the logarithm of the cardinality, we approximate the cost of query plans via sets of linear constraints and via auxiliary variables.

We must approximate cost functions since the cost of standard operators is usually not linear in the logarithm of input and output cardinalities. We can however choose the approximation precision by choosing the number of constraints and auxiliary variables. This allows in principle arbitrary degrees of precision. Also note that there are entire sub-domains of query optimization in which it is standard to approximate plan cost functions via linear functions [59, 73, 74, 139]. Approximating plan cost via linear function is therefore a widely-used approach.

Our goal here was to give a first intuition for how our transformation works and we have therefore considered join order alone and in a simplified setting. Later we show how to extend our approach for representing alternative operator implementations, complex cost models taking into account interesting orders and the evaluation cost of expensive predicates, or richer query languages.

We formally analyze our transformation in terms of the resulting number of constraints and variables. In our experimental evaluation, we apply the Gurobi MILP solver to query optimization problems that have been reformulated as MILP problems. We compare against a classical dynamic programming based query optimization algorithm on different query sizes and join graph structures. Our results are encouraging: the MILP approach often generates guaranteed near-optimal query plans after few seconds where dynamic programming based optimization does not generate any plans up to the timeout of one minute.

The original scientific contributions of this chapter are the following:

- We show how to reformulate query optimization as MILP problem.
- We analyze the problem mapping and express the number of variables and constraints as function of the query dimensions.
- We evaluate our approach experimentally and compare against a classical dynamic programming based query optimizer.

The remainder of this chapter is organized as follows. We discuss related work in Section 7.2. In Section 7.3, we introduce our formal problem model. Section 7.4 describes how we transform query optimization into MILP. We analyze how the size of the resulting MILP problem grows in the dimension of the original query optimization problem in Section 7.6. In Section 7.7, we experimentally evaluate an implementation of our MILP approach in comparison with a classical dynamic programming based query optimization algorithm.

## 7.2 Related Work

MILP representations have been proposed for many optimization problems in the database domain, including but not limited to multiple query optimization [54], index selection [108], materialized view design [148], selection of data samples [13], or partitioning of data for parallel processing [20]. In the areas of parametric query optimization and multi-objective parametric query optimization it is common to model the cost of query plans by linear functions that depend on unknown parameters [59, 73, 74, 139]. None of those prior publications formalizes however the join ordering and operator selection problem as MILP.

Query optimization algorithms can be roughly classified into exhaustive algorithms that formally guarantee to find optimal query plans and into heuristic algorithms which do not possess those formal guarantees. Exhaustive query optimization algorithms are often based on dynamic programming [116, 141, 100, 101]. We compare against such an approach in our experimental evaluation.

Our MILP-based approach to query optimization can be used as an exhaustive query optimization algorithm since we can configure the MILP solver to return a guaranteed-optimal solution. The MILP solver can however easily be configured to return solutions that are guaranteed near-optimal (i.e., the cost of the result plan is within a certain factor of the optimum) or to return the best possible plan within a given amount of time. This makes the MILP approach more flexible than typical exhaustive query optimization algorithms. Furthermore, MILP solvers posses the anytime property, meaning that they produce multiple plans of decreasing cost during optimization. The development of anytime algorithms for query optimization has recently been identified as a research challenge [34]. Transforming query optimization into MILP immediately yields anytime query optimization. Note that the anytime algorithms described in Chapter 3 cannot speed up traditional query optimization with one plan cost metric.

The parallelization of exhaustive query optimization algorithms (not to be confused with query optimization for parallel execution) is currently an active research topic [67, 68, 129, 145]. MILP solvers such as Cplex or Gurobi are able to exploit parallelism and transforming query optimization into MILP hence yields parallel query optimization as well. The development of parallel query optimizers for new database systems requires generally significant investments [129]; the amount of code to be written can be significantly reduced by using a generic solver as optimizer core.

Various heuristic and randomized algorithms have been proposed for query optimization [23, 31, 76, 131, 135, 134]. In contrast to many exhaustive algorithms, most of them possess the anytime property and generate plans of improving quality as optimization progresses. Those approaches can however not give any formal guarantees at any point in time about how far the current solution is from the optimum. MILP solvers provide upper-bounds during optimization on the cost difference between the cost of the current solution and the theoretical optimum. Such bounds can for instance be used to stop optimization once the distance reaches a threshold. Randomized algorithms do not offer that possibility and the returned solutions may be arbitrarily far from the optimum.

## 7.3 Model and Assumptions

Our notation is similar to the ones used in previous chapters. Nevertheless, we introduce notation from scratch to make the current chapter self-contained.

The goal of query optimization is to find an optimal or near-optimal plan for a given query. It is common to introduce new query optimization algorithms by means of simplified problem models. We also use a simple query and query plan model throughout most of the chapter while we discuss extensions to richer query languages and plan models as well.

In our simplified model, we represent a query as a set $Q$ of tables that need to be joined together with a set $P$ of binary predicates that connect the tables in $Q$ (extensions to nested queries, queries with aggregates, queries with projections, and queries with non-binary predicates will be discussed). For each binary predicate $p \in P$, we designate by $T_1(p), T_2(p) \in Q$ the two tables that the predicate refers to. Predicates can only be evaluated in relations in which both tables they refer to have been joined.

We assume in the simplified problem model that one scan and one binary join operator are available. As we consider binary joins, a query with $n$ tables requires $n-1$ join operations. A query plan is defined by the operands of those $n-1$ join operations, more precisely by the tables that are present in those operands. We consider left-deep plans. For left-deep query plans, the inner operand is always a single table; the outer operand is the result from the previous join (except for the outer operand of the first join which is a single table).

Query plans are compared according to their execution cost. The execution cost of a plan depends on the cardinality of the intermediate results it produces. We write $Card(t) \geq 1$ to designate the cardinality of table $t$ and $Sel(p) \in (0,1]$ to designate the selectivity of predicate $p$. We assume in the simplified model that the cardinality of the join between several tables, after having evaluated a set of join predicates, corresponds to the product of the table cardinalities and the predicate selectivities. We hence assume in the simplified model uncorrelated predicates while extensions to correlated predicates will be discussed. We generally assume that the execution cost of a query plan is the sum of the execution cost of all its operations. We will show how to represent various cost functions.

We translate the problem of finding a cost-minimal plan for a given query into a mixed integer linear programming problem (MILP). A MILP problem is defined by a set of variables (that can have either integer or continuous value domains), a set of linear constraints on those variables, and a linear objective function on those variables that needs to be minimized. A solution to a MILP is an assignment from variables to values from the respective domain such that all constraints are satisfied. An optimal solution minimizes the objective function value among

Table 7.1 – Variables for formalizing join ordering for left-deep query plans as integer linear program.

| **Symbol** | **Domain** | **Semantic** |
|---|---|---|
| $tio_{tj}/tii_{tj}$ | $\{0,1\}$ | If table $t$ is in outer/inner operand of $j$-th join |
| $pao_{pj}$ | $\{0,1\}$ | If $p$-th predicate can be evaluated on outer operand of $j$-th join |
| $lco_j$ | $\mathbb{R}$ | Logarithm of cardinality of outer operand of $j$-th join |
| $cto_{rj}$ | $\{0,1\}$ | If cardinality of outer operand of $j$-th join reaches $r$-th threshold |
| $co_j/ci_j$ | $\mathbb{R}_+$ | Approximated cardinality of outer/inner operand of $j$-th join |

all solutions.

## 7.4 Join Ordering Approach

The join ordering problem is usually solved by algorithms that are specialized for that problem and run inside the query optimizer. We adopt a radically different approach: we translate the join ordering problem into a MILP problem that we solve by a generic MILP solver.

MILP is an extremely popular formalism that is used to solve a variety of problems inside and outside the database community. By mapping the join ordering problem into a MILP formulation, we benefit from decades of theoretical research in the area of MILP as well as from solver implementations that have reached a high level of maturity. By linking query optimization to MILP, we make sure that query optimization will from now on indirectly benefit from all theoretical advances and refined implementations that become available in the MILP domain.

We explain in the following our mapping from a join ordering problem to a MILP. We describe the variables and constraints by which we represent valid join orders in Section 7.4.1. We show how to model the cardinality of join operands in Section 7.4.2. In Section 7.4.3 we associate plans with cost values based on the operand cardinalities.

Note that we introduce our mapping by means of a basic problem model in this section while we discuss extensions to the query language, plan space, and cost model in Section 7.5.

### 7.4.1 Join Order

A MILP program is characterized by variables with associated value domains, a set of linear constraints on those variables, and a linear objective function on those variables that needs to be minimized. Table 7.1 summarizes the variables that we require to model join ordering as MILP problem and Table 7.2 summarizes the associated constraints. We introduce them step-by-step in the following.

Table 7.2 – Constraints for join ordering in left-deep plan spaces.

| **Constraint** | **Semantic** |
|---|---|
| $\sum_t tio_{t0} = 1 / \forall j : \sum_t tii_{tj} = 1$ | Select one table for outer operand of first join/for all inner operands |
| $\forall j \forall t : tio_{tj} + tii_{tj} \leq 1$ | The tables in the join operands cannot overlap for the same join |
| $\forall j \geq 1 \forall t : tio_{tj} = tii_{t,j-1} + tio_{t,j-1}$ | Results of prior join are outer operand for next join |
| $\forall p \forall j : pao_{pj} \leq tio_{T_1(p)j}; pao_{pj} \leq tio_{T_2(p)j}$ | Predicates are applicable if both referenced tables are in outer operand |
| $\forall j : ci_j = \sum_t Card(t) tii_{tj}$ | Determines cardinality of inner operand |
| $\forall j : lco_j = \sum_t \log(Card(t)) tio_{tj} +$ $\sum_p \log(Sel(p)) pao_{pj}$ | Determines logarithm of outer operand cardinality, taking into account selected tables and applicable predicates |
| $\forall j \forall r : lco_j - cto_{rj} \cdot \infty \leq \log(\theta_r)$ | Activates threshold flag if cardinality reaches threshold |
| $\forall j : co_j = \sum_r cto_{rj} \delta\theta_r$ | Translates activated thresholds into approximate cardinality |

We start by discussing the variables and constraints that we need in order to represent valid left-deep query plans. Later we discuss the variables and constraints that are required to estimate the cost of query plans.

We represent left-deep query plans for a query $Q$ as follows. For the moment, we assume that only one join operator and one scan operator are available while we discuss extensions in Section 7.5. Under those assumptions, a query plan is specified by the join operands. We introduce a set of binary variables $tio_{tj}$ (short for *Table In Outer join operand*) with the semantic that $tio_{tj}$ is one if and only if query table $t \in Q$ appears in the outer join operand of the $j$-th join. We numerate joins from 0 to $j_{max}$ where $j_{max}$ is determined by the number of query tables. Analogue to that, we introduce a set of binary variables $tii_{tj}$ (short for *Table In Inner join operand*) indicating whether the corresponding table is in the inner operand of the $j$-th join.

The variables representing left-deep plans have binary value domains. Note that not all possible value combinations represent a valid left-deep plan. For instance, we could represent joins with empty join operands. Or we could build plans that join only a subset of the query tables and are therefore incomplete. We must impose constraints in order to restrict the considered value combinations to the ones representing valid and complete left-deep plans.

Left-deep plans are characterized by the particularity that the inner operand consists of only one table for each join. We capture that fact by the constraint $\sum_t tii_{tj} = 1$ which we need to introduce for each join $j$. A similar constraint restricts the table selections for the outer operand of the first join (join index $j = 0$) as only one table can be selected as initial operand. For the following joins (join index $j \geq 1$), the outer join operand is always the result of the previous join which is another characteristic of left-deep plans. This translates into the constraints $tio_{tj} = tii_{t,j-1} + tio_{t,j-1}$.

The latter constraint actually excludes the possibility that the same table appears in both operands of a join (since the result of the sum between $tii_{t,j-1} + tio_{t,j-1}$ cannot exceed the maximal value of one for $tio_{tj}$) except for the last join. We add the constraint $tio_{tj_{max}} + tii_{tj_{max}} \leq 1$ for the last join (and optionally for the other joins as well).

The number of joins is one less than the number of query tables. We join two (different) tables in the first join. After that, each join adds one new table to the set of joined tables since the outer operand contains all tables that have been joined so far, since the inner operand consists of one table, and since inner and outer join operands do not overlap. As a result, we can only represent complete query plans that join all tables.

We could have chosen a different representation of query plans with less variables. The problem is that we need to be able to approximate the cost of query plans based on that representation using linear functions. Our representation of query plans might at first seem unnecessarily redundant but it allows to impose the constraints that we discuss next. Also note that MILP solvers typically try to eliminate unnecessary variables and constraints in preprocessing steps. This makes it less important to reduce the number of variables and constraints at the cost of readability.

**Example 15.** *We illustrate the representation of left-deep query plans for the join query $R \bowtie S \bowtie T$. Answering the query requires two join operations. Hence we introduce six variables $tio_{tj}$ for $t \in \{R, S, T\}$ and $j \in \{0, 1\}$ to represent outer join operands and six variables $tii_{tj}$ to represent inner join operands. The join order $(R \bowtie S) \bowtie T$ is for instance represented by setting $tio_{R0} = tii_{S0} = 1$ and $tio_{R1} = tio_{S1} = tii_{T1} = 1$ and setting the other variables representing join operands to zero. This assignment satisfies the two constraints that restrict inner operands to single tables (e.g., $\sum_{t \in \{R,S,T\}} tii_{t1} = 1$ for the second join), it satisfies the constraint restricting the outer operand in the first join to a single table ($\sum_{t \in \{R,S,T\}} tio_{t0} = 1$), and it satisfies the constraints making the outer operand of the second join equal to the union of the operands in the first join (e.g., $tio_{R1} = tio_{R0} + tii_{R0}$).*

### 7.4.2 Cardinality

Our goal is to find query plans with minimal cost and hence we must associate query plans with a cost value. The execution cost of a query plan depends heavily on the cardinality of intermediate results. We need to represent the cardinality of join operands and join results in order to calculate the cost of query plans. Inner operands consist always of a single table and calculating their cardinality is straight-forward: designating by $ci_j$ (short for *Cardinality of Inner operand*) the cardinality of the inner operand of join number $j$, we simply set $ci_j = \sum_t tii_{tj} Card(t)$ where $Card(t)$ is the cardinality of table $t$.

Calculating cardinality for outer join operands is however non-trivial as we can only use linear constraints: the cardinality of a join result is usually estimated as the product of the cardinalities of the join operands times the selectivity of all predicates that are applied during the join. The product is a non-linear function and does not directly translate into linear constraints.

We circumvent that problem via the following trick. While cardinality is actually defined as the product of table cardinality values and predicate selectivity values, we represent the logarithm of the cardinality instead and the logarithm of a product is the sum of the logarithms of the factors. More formally, given a set $T \subseteq Q$ of query tables such that the set of predicates $P$ is applicable to $T$ (i.e., for each binary predicate in $P$ the two tables it refers to are included in $T$) and designating by $Card(t)$ for $t \in T$ the cardinality of table $t$ and by $Sel(p)$ the selectivity of predicate $p \in P$, a cardinality estimate is given by $\prod_{t \in T} Card(t) \cdot \prod_{p \in P} Sel(p)$ and the logarithm of the cardinality estimate is $\sum_{t \in T} \log(Card(t)) + \sum_{p \in P} \log(Sel(p))$ which is a linear function.

We introduce the set of variables $lco_j$ (short for *Logarithmic Cardinality of Outer operand*) which represents the logarithm of the cardinality of the outer operand of the $j$-th join. The aforementioned linear formula for calculating the logarithm of the cardinality depends on the selected tables as well as on the applicable predicates. The selected tables are directly given in the variables $tio_{tj}$. We introduce additional binary variables to represent the applicable predicates: variable $pao_{pj}$ (short for *Predicate Applicable in Outer join operand*) captures whether predicate $p$ is applicable in the outer operand of the $j$-th join. We currently consider only binary predicates (we discuss extensions later) and as the inner operands consist of single tables, we do not need to introduce an analogue set of predicate variables for the inner operands.

We denote by $T_1(p)$ and $T_2(p)$ the first and the second table that predicate $p$ refers to. A predicate is applicable to an operand whose table set $T$ contains $T_1(p)$ and $T_2(p)$. We make sure that predicates cannot be applied if one of the two tables is missing by adding for each predicate $p$ and each join $j$ a pair of constraints of the form $pao_{pj} \leq tio_{T_1(p)}$ and $pao_{pj} \leq tio_{T_2(p)}$. We currently assume that predicate evaluations do not incur any cost while extensions are discussed later. Under this assumption, applying a predicate has only beneficial effects as it reduces the cardinality of intermediate results and therefore the cost of the following joins. This means that we only need to introduce constraints preventing the solver from using predicates that are inapplicable but we do not need to add constraints forcing the evaluation of predicates explicitly.

Using the variables capturing the applicability of predicates, we can now write the logarithm of the join operand cardinalities. For outer join operands, we set

$$lco_j = \sum_t \log(Card(t))tio_{tj} + \sum_p \log(Sel(p))pao_{pj}$$

and thereby take into account table cardinalities as well as predicate selectivities.

Unfortunately, the cost of most operations within a query plan is not linear in the logarithm of the cardinality values. In the following, we show how to transform the logarithm of the cardinality values into an approximation of the raw cardinality values. This allows to write cost functions that are linear in the cardinality of their input and output. This is sufficient for many but not for all standard operations. Similar techniques to the ones we describe in the following can however be used to represent for instance log-linear cost functions as we describe in more detail in Section 7.4.3.

We must transform the logarithm of the cardinality into the cardinality itself. This is not a linear transformation and hence we resort to approximation. We assume that a set $\Theta = \{\theta_r\}$ of cardinality threshold values has been defined for integer indices $r$ with $0 \leq r \leq r_{max}$. In addition, we introduce a set of binary variables $cto_{rj}$ (short for *Cardinality Threshold reached by Outer operand*) that indicate for each join $j$ and each cardinality threshold value $\theta_r$ whether the cardinality of the outer operand reaches the corresponding threshold value. If threshold $\theta_r$ is reached then the corresponding threshold variable $cto_{rj}$ must take value one and otherwise value zero. To guarantee that the previous statement holds, we introduce constraints of the form $lco_j - cto_{rj} \cdot \infty \leq \log(\theta_r)$ for each join $j$ where $\infty$ is in practice a sufficiently large constant such that the constraint can be satisfied by setting the threshold variable $cto_{rj}$ to one. We do not explicitly enforce that the threshold variable is set to zero in case that the threshold is not reached. The constraints that we introduce next make however sure that the cardinality estimate and therefore the cost estimate increase with every threshold variable that is set to one. Hence the solver will set the threshold variables to zero wherever it can.

Based on the threshold variables, we can formulate a linear approximation for the raw cardinality. We introduce the set of variables $co_j$ representing the raw cardinality of the outer operand of the $j$-th join and set $co_j = \sum_r cto_{rj}\delta\theta_r$ where the values $\delta\theta_r$ are chosen appropriately such that if threshold variables $cto_{0j}$ up to $cto_{mj}$ are set to one for some specific join $j$ then the cardinality variable $co_j$ takes a value between $\theta_m$ and $\theta_{m+1}$ (assuming that thresholds are indexed in ascending order such that $\forall r : \theta_r < \theta_{r+1}$). We can for instance set $\delta\theta_r = \theta_r - \theta_{r-1}$ for $r \geq 1$ and $\delta\theta_0 = \theta_0$.

**Example 16.** *We illustrate how to calculate join operand cardinalities and continue the previous example with join query $R \bowtie S \bowtie T$. We have two joins and introduce therefore four variables ($ci_0$, $ci_1$, $co_0$, and $co_1$) representing operand cardinalities. Assume that tables $R$, $S$, and $T$ have cardinalities 10, 1000, and 100 respectively. We calculate the cardinality of the two inner join operands by summing over the variables indicating the presence of a table in an inner operand, weighted by the cardinality values (e.g., $ci_0 = 10tii_{R0} + 1000tii_{S0} + 100tii_{T0}$). The cardinality of the outer operands can depend on predicates. Assume that one predicate $p$ is defined between* *tables R and S. We introduce two variables, $pao_{p0}$ and $pao_{p1}$, indicating whether the predicate can be evaluated in the outer operand of the corresponding join. Predicates can be evaluated if both referenced tables are in the corresponding operand. We introduce four constraints (e.g., $pao_{p0} \leq tio_{R0}$ and $pao_{p0} \leq tio_{S0}$) forcing the value of the predicate variable to zero if at least one of the tables is not present. We introduce two variables storing the logarithm of the outer operand cardinality: $lco_0$ and $lco_1$. We assume that the selectivity of p is 0.1. Then the logarithmic cardinality for the first outer join operand is given by $lco_0 = 1pao_{R0} + 3pao_{S0} + 2pao_{T0} - 1pao_{p0}$, assuming that the logarithm base is 10. To simplify the example, we assume that only two cardinality thresholds are considered: $\theta_0 = 10$, and $\theta_1 = 1000$. We introduce four variables $cto_{rj}$ with $r \in \{0,1\}$ and $j \in \{0,1\}$ indicating whether the cardinality of the outer join operand reaches each threshold for the first or second join. Each threshold variable is constrained by one constraint (e.g., $lco_0 - \infty \cdot cto_{0,0} \leq 1$). Now we define the cardinality of the outer join operands by constraints such as $co_0 = 10cto_{0,0} + (1000-10)cto_{1,0}$. This provides a lower bound for the true cardinality. If we know for instance that cardinality values are upper-bounded by 100000 due to the query properties, we can also set $co_0 = 100cto_{0,0} + (10000-100)cto_{1,0}$. Then the difference between true and approximate cardinality is at most one order of magnitude.*

### 7.4.3 Cost

Now we can for instance sum up the cardinalities over all intermediate results ($\sum_{j\geq 1} cio_j$) and thereby obtain a simple cost metric that is equivalent to the $C_{out}$ cost metric introduced by Cluet and Moerkotte [41]. Join orders minimizing that cost metric were shown to minimize cost according to the cost formulas of some of the standard join operators as well [41]. We will however show in the following how the cost of all standard join operators, namely hash join, sort-merge join, and block nested loop join, can be modeled directly.

The standard cost formula for a hash join operation is based on the number of pages that the two input operands consume on disk. We designate by $pgo_j$ the number of disk pages consumed by the outer operand of join number $j$ and $pgi_j$ is the analogue value for the inner operand. If a hash join operator is used for the join then its cost is given by $3 \cdot (pgo_j + pgi_j)$. This is a linear formula but we must calculate the size of the operands in disc pages.

The byte size of an intermediate result, and therefore the number of consumed disk pages, depends not only on the cardinality but also on the columns that are present. For the moment, we make the simplifying assumption that each tuple has a fixed byte size. We show how to relax that restriction in the next section. Under this simplifying assumption, we can however express the disk pages of the outer operands as $pgo_j = \lceil co_j \cdot tupSize/pageSize \rceil$ where $tupSize$ is the fixed byte size per tuple and $pageSize$ the number of bytes per disk page. Factor $tupSize/pageSize$ is a constant due to our simplifying assumption and hence we can set $pgo_j = co_j \cdot tupSize/pageSize$ to obtain the approximate number of disk pages. Alternatively, we could write $pgo_j = \sum_r \lceil \theta_r \cdot tupSize/pageSize \rceil (cto_{jr} - cto_{j,r+1})$ and approximate it using the threshold variables (the expression $(cto_{jr} - cto_{j,r+1})$ yields value one only for the threshold variable with the highest threshold that is still set to one). Note that the factors of the form $\lceil \theta_r \cdot tupSize/pageSize \rceil$ are constants. The second version has the advantage that we can explicitly control the approximation precision for $pgo_j$ by tuning the number of thresholds. The disc pages for the inner operands can be obtained in a simplified way as each inner operand consists of only one table: we simply set $pgi_j = \sum_t tii_{tj} \lceil Card(t) \cdot tupSize/pageSize \rceil$.

The cost of sort-merge join operators can be approximated in a similar way. We assume here that both inputs must be sorted while we generalize in the next subsection. If both input operands need to be sorted first then the join cost is given by $2pgo_j \lceil \log(pgo_j) \rceil + 2pgi_j \lceil \log(pgi_j) \rceil + pgo_j + pgi_j$. We have already shown how to obtain the number of disc pages $pgo_j$ and $pgi_j$. The log-linear numbers of disc pages, $pgo_j \log(pgo_j)$ and $pgi_j \log(pgi_j)$, can be obtained in a similar way. We use the cardinality thresholds for the outer operand and simply sum over tables for the inner operand.

The cost function for the block nested loop join is given by $\lceil pgo_j/buffer \rceil \cdot pgi_j$ where $buffer$ is the amount of buffer space dedicated to the outer operand. We assume here that pipelining is used while the generalization is straightforward. There are several options for approximating that cost function with linear constraints. We can approximate the join cost function by omitting the ceiling operator and obtain $pgo_j/buffer \cdot pgi_j$. Similar to how we calculated the cardinality of the outer operands, we can switch to a logarithmic representation and write the logarithm of the join cost as $\log(pgo_j) + \log(pgi_j) - \log(buffer)$. Then we can transform the logarithm of the join cost into the raw join cost value using a set of newly introduced threshold variables.

Another idea is to exploit the specific shape of the inner join operands. As only one table is selected for the inner join operand, we can express join cost by the formula $\sum_t tii_{tj} \cdot pages(t) \cdot blocks_j$ where $pages(t) = \lceil Card(t) \cdot tupSize/pageSize \rceil$ designates the disk page size of table $t$ and $blocks_j = \lceil pgo_j/buffer \rceil \approx pgo_j/buffer$ is the number of iterations of the outer loop executed by the block nested loop join. This is a weighted sum over products between a binary variable (the variables $tii_{tj}$ indicating whether table $t$ was selected for the inner operand of join number $j$) and a continuous variable (the variables $blocks_j$). This formula is hence not directly linear but the product between a binary variable and a continuous variable can be expressed by introducing one auxiliary variable and a set of constraints [26]. The only condition for this transformation is that the continuous variable is non-negative and upper-bounded by a constant. Both is the case (note that we generally only model a bounded cardinality range which implies also an upper bound on the number of loop iterations). The advantage of the second representation is that we only need to introduce a number of variables and constraints that is linear in the number of tables (instead of linear in the number of thresholds like for the first possibility).

We have seen that join orders, the cardinality of intermediate results, and the cost of join operations according to standard cost formulas can all be represented as MILP. In the next section we introduce several extensions of the problem model that we used so far.

## 7.5 Extensions

We introduced our mapping for query plans by means of a basic problem model that focuses on join order. We discuss extensions of the query language, of the query plan model, and of the cost model in this section.

Note that not all proposed extensions are necessary in each scenario: the basic model introduced in the last section allows for instance to find join orders which minimize the sum of intermediate result sizes. Such join orders are optimal according to many standard operator cost functions [41]. It is therefore in many scenarios possible to obtain good query plans based on the join order that was calculated using the basic model. To transform a join order into a query plan, we choose optimal operator implementations based on the cardinality of the join operands, we evaluate predicates as early as possible (predicate push-down), and we project out columns as soon as they are not required anymore.

An alternative is to let the MILP solver make some of the decisions related to projection, predicate evaluation, and join operator selection. We show how this can be accomplished if desired. In addition, we discuss extensions of the cost and query model.

In Section 7.5.1, we discuss how to represent n-ary predicates, correlated predicates, and predicates that are expensive to evaluate. We show how to handle projections in Section 7.5.2 and in Section 7.5.3 we show how the MILP solver can choose between different operator implementations. We show how to handle interesting orders and other intermediate result properties in Section 7.5.4. In Section 7.5.5, we finally discuss how we can extend our approach to handle queries with aggregates and nested queries.

We sketch out the following extensions relatively quickly due to space restrictions. They use however similar ideas as we applied in the last section. Our goal is less to provide a detailed model for each possible scenario but rather to demonstrate that the MILP formalism is flexible enough to cover the most relevant aspects of query optimization.

### 7.5.1 Predicate Extensions

So far we have considered binary predicates. We show how n-ary predicates can be modeled. Let $p$ be an n-ary predicate. N-ary predicates refer to n tables and we designate by $T_1(p)$ to $T_n(p)$ the tables on which $p$ is evaluated. All tables that $p$ refers to must be present in the operands in which $p$ is evaluated. If $pao_{pj}$ indicates whether predicate $p$ can be evaluated in the outer operand of the $j$-th join then we must introduce constraint $pao_{pj} \leq tio_{T_i(p)j}$ for each join and each $i \in \{1, \dots, n\}$. This forces variables $pao_{pj}$ to zero if at least one table is not present. Note that we must introduce analogue predicate variables for the inner operands for all unary predicates.

In our basic model, we assume that predicates are uncorrelated. Then the accumulated selectivity of a predicate group corresponds always to the product of the selectivity values of the single tables. In reality this is not always the case, even if it is a common simplification to assume uncorrelated predicates. Assume that there is a correlated group $P_{cor}$ of predicates such that the accumulated selectivity of all predicates in $P_{cor}$ differs significantly from their selectivity product. Then we introduce a new predicate $g$ that represents the correlated predicate group. The selectivity $Sel(g)$ is chosen in a way such that $Sel(g)\prod_{p\in P_{cor}} Sel(p)$ yields the correct selectivity, taking correlations into account. So the selectivity of $g$ *corrects* the erroneous selectivity that is based on the assumption of independent predicates.

Now we just need to make sure that the predicate variable associated with $g$ is set to one in all operands in which all predicates from $P_{cor}$ are selected but not otherwise. We force $pao_{gj}$ to one if all correlated predicates are present by requiring $pao_{gj} \geq 1 - |P_{cor}| + \sum_{p\in P_{cor}} pao_{pj}$. We force $pao_{gj}$ to zero if at least one of the correlated predicates is not activated by introducing $n$ constraints of the form $pao_{gj} \leq pao_{pj}$ for $p \in P_{cor}$. No other constraints need to be introduced for $pao_{gj}$ but terms including $pao_{gj}$ must be included in all expressions representing cardinality, byte size, etc.

So far we have assumed that predicate evaluation is not associated with cost. We constrained the variables $pao_{pj}$ only to zero if required tables are not in the operand. We did not explicitly force them to one at any point since, as they reduce cardinality, their evaluation reduces cost and the MILP solver will generally choose to evaluate them as early as possible.

This model is not always appropriate. If predicate evaluations are expensive then it can be preferable to postpone their evaluation [35, 70, 84]. The predicate-related variables $pao_{pj}$ influence the cardinality estimates of join operands. They capture whether the corresponding predicate *was* already evaluated as otherwise it cannot influence cardinality. We cannot use those variables directly to incorporate the cost of predicate evaluations. The effect on cardinality of having evaluated a predicate once will persist for all future operations. The evaluation cost needs however only to be payed once. We introduce additional variables $pco_{pj}$ (short for *Predicate evaluation Cost for Outer operand*) and set $pco_{pj} = pao_{p,j+1} - pao_{p,j}$. Intuitively, the predicate was evaluated in the current join if it is evaluated in the input to the next join but not in the input of the current join. The sum $\sum_j pco_{pj}co_j$ yields the evaluation cost associated with predicate $p$ (we can additionally weight by a factor that represents predicate evaluation cost per tuple). This is not a linear function as we multiply variables. We have however a product between a binary variable and a continuous variable again. As before, we can transform such expressions into a set of linear constraints and a new auxiliary variable [26].

Now that evaluation of predicates is not automatically desirable anymore, we must introduce additional constraints making sure that all predicates are evaluated at the end of query execution. Designating by $j_{max}$ the index of the last join, we simply set $pao_{p,j_{max}+1} = 1$ by convention. This means that each predicate that was not evaluated before the last join must be evaluated during the last join since $pco_{pj_{max}} = 1 - pao_{pj_{max}}$. We finally introduce constraints making sure that no predicate is initially evaluated and we introduce constraints making sure that an evaluated predicate remains evaluated. The latter constraints are in fact optional since additional predicate evaluations increase the cost. Depending on the solver implementation, it can nevertheless be beneficial to add such constraints to reduce the search space size.

### 7.5.2 Projection

Our cost formulas have so far been based on cardinality alone as we have assumed a constant byte size per tuple. This is of course a simplification and we must in general take into account the columns that we project on and their byte sizes. We designate by $L$ the set of columns over all query tables. By $Byte(l)$ we denote the number of bytes per tuple that column $l \in L$ requires. We introduce one variable $clo_{jl}$ (short for *CoLumn in Outer operand*) for each join $j$ and each column $l \in L$ to indicate whether column $l$ is present in the outer operand of join $j$ (and analogue variables for the inner operands). Then a refined formula for the estimated number of bytes consumed by the outer operand is $co_j \cdot \sum_{l \in L} clo_{jl} Byte(l)$. This is the sum over products between a constant ($Byte(l)$), a binary variable ($clo_{jl}$), and a continuous variable that takes only non-negative values ($co_j$). This formula can be expressed using only linear constraints using the same transformations that we used already before [26]. Special rules apply for the inner operand again: for the inner operand, we can estimate the byte size (or any derived measure such as the number of disc pages) by summing over the column variables, weighted by the column byte size as well as by the cardinality of the table that the column belongs to.

We must still constrain the variables $clo_{jl}$ to make sure that only valid query plans can be represented. First of all we must connect columns to their respective tables. If the table associated with a column is not present then the column cannot be present either in a given operand. If column $l$ is associated with table $t$ then the constraint $clo_{jl} \leq tio_{tj}$ forces the column variable to zero if the associated table is not present. Not selecting any columns would be the most convenient way for the optimizer to reduce plan costs. To prevent this from happening, we must enforce that all columns that the query refers to are in the final result. Also, we must enforce that all columns that predicates refer to are present once they are evaluated. We introduced variables indicating the immediate evaluation of a predicate during a specific join. Those are the variables that need to be connected to the columns they require via corresponding constraints. We must also make sure that a column cannot reappear in later joins after it has been projected out (otherwise that would be a convenient way of reducing intermediate result sizes while still satisfying the constraints requiring certain columns in the final result). Introducing constraints of the form $clo_{jl} \geq clo_{j+1,l}$ satisfies that requirement.

### 7.5.3 Choosing Operator Implementations

We have already discussed the cost functions of different join operator implementations in the last section. So far we have however assumed that only one of those cost functions is used to calculate the cost for all joins. This allows to select optimal operator implementations after a good join order, minimizing intermediate result sizes, has been found. We can however also task the MILP solver to pick operator implementations as we outline in the following.

Denote by $I$ the set of join operator implementations. We have shown how to calculate join cost for each of the standard join operators. We can introduce a variable $pjc_{ji}$ (short for *Potential Join Cost*) for each join $j$ and for each operator implementation $i \in I$ representing the cost of the join if that operator is used. We use the term *potential* since whether that cost is actually counted depends on whether or not the corresponding operator implementation is selected.

We introduce binary variables $jos_{ji}$ (short for *Join Operator Selected*) to indicate for each operator implementation $i$ and join $j$ whether the operator was used to realize the join. We require that exactly one implementation is selected for each join as expressed by the constraint $\sum_i jos_{ji} = 1$ that we must introduce for each join. Having the potential cost for each join operator as well as information on which operator is selected, we can for each operator calculate the actual join cost $ajc_{ji}$. The actual join cost associated with one specific operator implementation is *zero* if that operator is not selected. Otherwise (if that operator is selected) the actual join cost corresponds to the potential join cost. We have the following relationship between potential and actual join cost $ajc_{ji} = jos_{ji} \cdot pjc_{ji}$. Here we multiply a binary with a non-negative continuous variable and can apply the same linearization as before [26]. The sum over the actual join cost variables over all operator implementations yields the cost of each join operation.

### 7.5.4 Intermediate Result Properties

Alternative join operator implementations can sometimes produce intermediate results with different physical properties (while the contained data remains the same over all alternative implementations). Tuple orderings are perhaps the most famous example [116]. If tuples are produced in an interesting order then the cost of successive operations can be reduced (e.g., the sorting stage can be dropped for a sort-merge join). Also, the distinction whether an intermediate result is written to disc or remains in main memory is a physical property of that result and influences the cost of successive operations.

Assume that we consider a set $X$ of relevant intermediate result properties. Then we can introduce a binary variable $ohp_{jx}$ (short for *Outer operand Has Property*) indicating whether the outer operand of the $j$-th join has property $x$. Property $x$ could for instance represent the fact that the corresponding result is materialized. Property $x$ could also represent one specific tuple ordering.

The properties constrain the operator implementations that can be applied for the next join. We could for instance introduce one operator implementation representing a pipelined block nested loop join while another operator implementation represents a block nested loop join without pipelining. The applicability of the pipelined join would have to be restricted based on whether or not the corresponding input remains in memory. If implementation $i$ requires property $x$ in the outer join operand in order to become applicable then we can impose the constraint $jos_{ji} \leq ohp_{jx}$ to express that fact.

Operators such as the sort-merge join can be decomposed into different sub-operators (e.g., sorting the outer operand, sorting the inner operand, merging). This avoids having to introduce a new variable for each possible combination of situations (e.g., outer operand sorted and inner operand sorted, outer operand sorted but inner operand not sorted, etc.).

Whether an intermediate result has a certain physical property is determined by the operator which produces the result (and possibly by properties of the input to the producing operation). If a subset $\widetilde{I} \subseteq I$ produces results with a certain property $x$ then we can set $ohp_{j+1,x} = \sum_{i \in \widetilde{I}} jos_{ij}$. As only one of the operators is selected, the aforementioned constraint is valid and sets the left expression either to zero or to one. Certain properties such as interesting orders might be provided automatically by certain tables (if the data on disk has that order). Then we need additional constraints to connect properties to tables.

In summary, we have shown that all of the most important aspects of query optimization can be represented in the MILP formalism.

### 7.5.5 Extended Query Languages

We have already implicitly discussed several extensions to the query language in this section. We discussed how non-binary predicates and projection are supported. This gives us a system handling select-project-join (SPJ) queries.

It is generally common to introduce query optimization algorithms using SPJ queries for illustration. There are however standard techniques by which an optimization algorithm treating SPJ queries can be extended into an algorithm handling richer query languages.

The seminal paper by Selinger [116] describes how a complex SQL statement containing nested queries can be decomposed into several simple query blocks that use only selection, projection, and joins; the join order optimization algorithm is applied to each query block separately. Later, the problem of unnesting a complex SQL statement containing aggregates and sub-queries into simple SPJ blocks has been treated as a research problem on its own; corresponding publications focus on the unnesting algorithms and use join order optimization algorithms as a sub-function (e.g., [103]).

## 7.6 Formal Analysis

State-of-the art MILP solvers use a plethora of heuristics and optimization algorithms which makes it hard to predict the run time for a given MILP instance. It is however a reasonable assumption that optimization time tends to increase in the number of variables and constraints, even if preprocessing steps are sometimes able to eliminate redundant elements. The assumptions that we make here are supported by the experimental results that we present in the next section: we see a strong (even if not perfect) correlation between the number of variables and constraints and the MILP solver performance.

For the aforementioned reasons, we study in the following how the asymptotic number of variables and constraints in the MILP grows in the dimensions of the query optimization problem from which it was derived. We denote in the following by $n = |Q|$ the number of query tables to join and by $m = |P|$ the number of predicates. By $l = |\Theta|$ we denote the number of thresholds that are used to approximate cardinality values. The following theorems refer to the basic problem model that was presented in Section 7.4.

**Theorem 35.** *The MILP has $O(n \cdot (n + m + l))$ variables.*

*Proof.* Give $n$ tables to join, each complete query plan has $O(n)$ joins. We require $O(n)$ binary variables per join to indicate which tables form the join operands, we require $O(m)$ binary variables per operand to indicate which predicates can be evaluated, and we require $O(l)$ continuous variables per operand to calculate cardinality estimates. □

**Theorem 36.** *The MILP has $O(n \cdot (n + m + l))$ constraints.*

*Proof.* For each join operand we need $O(n)$ constraints to restrict table selections, $O(m)$ constraints to restrict predicate applicability, and $O(l)$ constraints to force the threshold variables to the right value. □

## 7.7 Experimental Evaluation

Using existing MILP solvers as base for the query optimizer reduces coding overhead and automatically yields parallelized anytime query optimization due to the features of typical MILP solvers. In this section, we compare the performance of a MILP based optimizer to a classical dynamic programming based query optimization algorithm.

We describe and justify our experimental setup in Section 7.7.1 and discuss our results in Section 7.7.2.

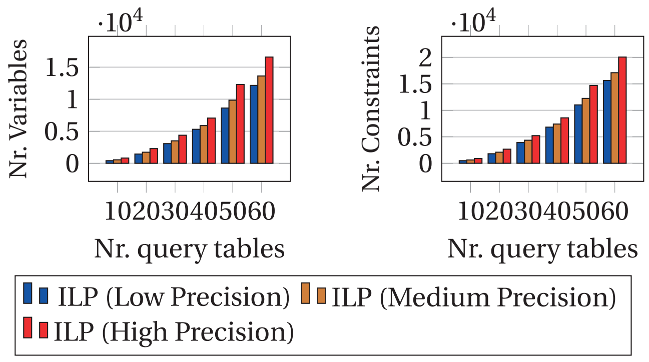


Figure 7.1 – Median number of variables and constraints of a MILP problem representing the optimization of one query.

### 7.7.1 Experimental Setup

We implemented a prototype of the MILP based optimizer that was introduced in the last sections. We transform query optimization problems into MILP problems and use the Gurobi[4] solver in version 5.6.3 to find optimal or near-optimal solutions to the resulting MILP problems. The MILP solution is read out and used to construct a corresponding query plan.

We compare this approach against the classical dynamic programming algorithm by Selinger [116]. Dynamic programming algorithms are very popular for exhaustive query optimization [100, 101] and are for instance used inside the optimizer of the Postgres database system[5].

We compare the two aforementioned algorithms on randomly generated queries. We generate queries according to the method proposed by Steinbrunn et al. [131] which is widely used to benchmark query optimization algorithms [131, 31, 139]. We generate queries of different sizes (referring to the number of tables to join) and with different join graph structures (chain graphs, star graphs, and cycle graphs [131]). We allow cross products which increases the search space size significantly compared to the case without cross products [106].

We assume that hash joins are used and search the optimal join order. The MILP approach approximates the byte sizes of the intermediate results and therefore the cost of join operations. We evaluate three configurations of our algorithm that differ in the precision by which they approximate cardinality (higher approximation precision requires more MILP variables and constraints). Our first configuration offers high precision and approximates cardinality with a tolerance of factor 3. Our second configuration reduces approximation precision and has a tolerance factor of 10. Our third configuration reduces approximation precision further and has tolerance factor 100. Our most precise configuration uses 60 threshold variables per intermediate result up to 40 table joins and 100 threshold variables per result for queries joining 50 and 60 tables. At the other side of the spectrum is the low-precision configuration which uses 15 threshold variables per result for up to 40 tables and 25 variables for more than

[4]http://www.gurobi.com/
[5]http://www.postgresql.org/

40 tables.

We compare algorithms by the quality of the plans that they produce after a certain amount of optimization time. We allow up to 60 seconds of optimization time and compare the output generated by all algorithms in regular time intervals. The high amount of optimization time seems justified since we compare the algorithms also on very large queries. All compared algorithms need significantly less time than 60 seconds to produce optimal plans for small queries. Investing 60 seconds into optimization can however be well justified if queries are executed on big data where choosing a sub-optimal plan can have devastating consequences [129].

During the 60 seconds of optimization time, we compare optimization algorithms in regular intervals according to the following criterion. We compare them based on the factor by which the cost of the best plan found so far is higher than the optimum at most. MILP solvers calculate such bounds based on the integrality gap. The classical dynamic programming algorithm is not an anytime algorithm but after its execution finishes, the produced plan is optimal and hence the optimality factor is one.

We *do not* compare algorithms based on the cost overhead that the generated plans have compared to the optimum. Instead, we compare them based on an *upper bound* on the relative cost overhead that the algorithm can formally guarantee at a certain point in time. The actual cost overhead is only known in hindsight after optimization has finished (and for some of the query sizes we consider, calculating the truly optimal query plans would cause high computational overheads). The upper bound that we use as criterion is the only value that is known at optimization time and therefore the only value on which termination decisions can be based on for instance (e.g., we could terminate optimization once the query optimizer is certain that the current plan is not more expensive than the optimum by more than factor 2).

The comparison criterion that we use excludes any randomized or heuristic query optimization algorithms [23, 31, 76, 131, 135, 134] from our experimental evaluation: such algorithms cannot give any formal guarantees on the optimality of the produced plans. They cannot even give upper bounds on the relative cost overhead of the generated plans.

Our algorithms (for the MILP approach: the part that transforms query optimization into MILP) are implemented in Java 1.7. The experiments were executed using the Java HotSpot(TM) 64-Bit Server Virtual Machine version on an iMac with i5-3470S 2.90GHz CPU and 16 GB of DDR3 RAM.

### 7.7.2 Experimental Results

We start by analyzing the size of the generated MILP problems. Figure 7.1 shows the number of constraints and variables. We show results for queries with a star-shaped join graph structure while the results for chain and cycle graph structures differ only marginally (the only difference is that cycle graphs require one additional predicate variable per intermediate result compared to star graphs). The ILP configuration with higher approximation precision requires in all cases more variables and constraints. For all configurations, the number of variables and constraints increases with increasing number of query tables.

Figure 7.2 shows performance results for left-deep plans. We allow cross product joins. The experimental setup was explained and justified in Section 7.7.1. The figure shows median values for 20 randomly generated queries. For 10 query tables, all compared algorithms find the optimal plan very quickly. For 20 query tables, the dynamic programming approach already takes more than six seconds in average to find the optimal plan while the MILP approach is faster. With 20 query tables we are reaching the limit of what is usually considered practical by dynamic programming algorithms. Also note that we allow cross product joins which increases the size of the plan space significantly.

For higher numbers of query tables, up to 60, the dynamic programming approach does not return any plan within one minute of optimization time. Note that increasing the number of tables by 10 increases the number of table sets that the dynamic programming approach must consider by factor $2^{10} = 1024$. It is therefore not surprising that this algorithm is not able to optimize queries with 30 tables and more.

All configurations of the MILP approach find optimal or at least guaranteed near-optimal plans for up to 40 tables, often already after a few seconds. For 50 and 60 table joins, all MILP configurations are able to find plans quickly for star join graphs. For cycle graphs, the low-precision configuration finds still optimal plans up to 60 tables while the medium-precision configuration finds near-optimal plans. Both configurations find optimal plans for 50 tables and chain graphs while this is not possible for queries with 60 tables and a chain graph structure. This means that optimization of chain and cycle queries seems to be more challenging for MILP approaches than optimization of star queries. Note that star queries are more difficult to optimize when excluding cross products and applying dynamic programming [106]; for MILP approaches it is apparently the opposite.

We conclude that the MILP approach does not only match but even outperforms traditional exhaustive query optimization algorithms for left-deep plan spaces by a significant margin.

## 7.8 Conclusion

Basing newly developed query optimizers on existing MILP solver implementations reduces the size of the optimizer code base and allows to benefit from features such as parallelization and anytime behavior that those solvers encapsulate.

We have demonstrated how to transform query optimization into MILP. Our experimental results show that MILP approaches can outperform traditional dynamic programming approaches significantly.

Generally it should be noted that the experimental results in this chapter are only snapshots and not intrinsic to the proposed mapping: as new MILP solver generations appear, the performance of our MILP based approach is likely to improve further without having to adapt the mappings.

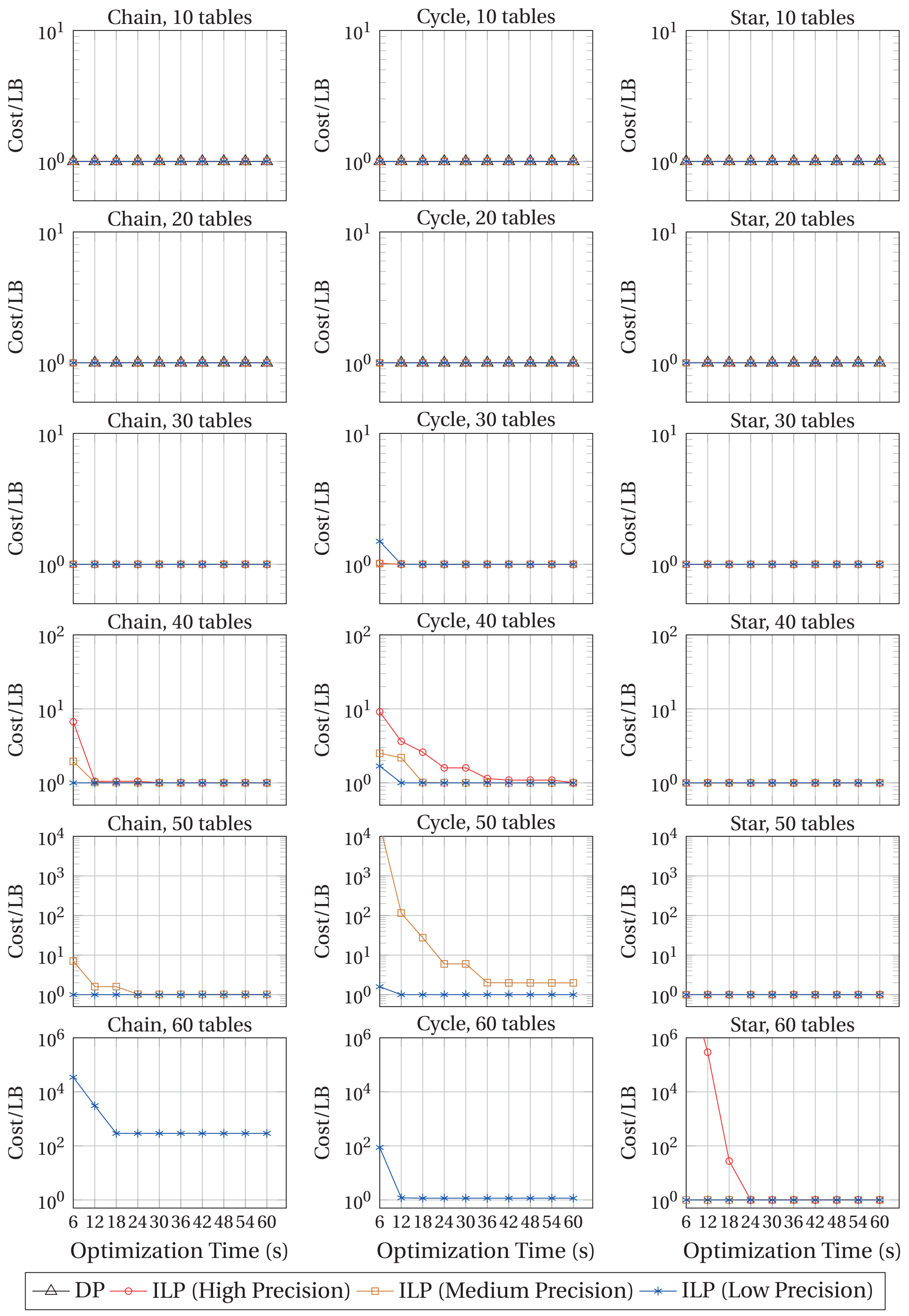


Figure 7.2 – Comparing dynamic programming based optimizer versus integer linear programming for left-deep query plans.

# 8 Quantum Computing

In the last chapter, we have seen how to solve query optimization by leveraging software solvers. In this chapter, we will see how to solve a query optimization variant using a very specific hardware solver: the D-Wave adiabatic quantum annealer. This device uses quantum effects to solve NP-hard optimization problem. We obtained access to such a device, located at NASA Ames Research Center in California, by a research grant. In order to use that device, we must again transform query optimization into a different representation: we must transform problem instances into strength values of magnetic fields on and between qubits. This is the input format accepted by the quantum annealer.

In this chapter, we will see a corresponding transformation method. We will analyze that transformation in terms of how the number of required qubits grows asymptotically in the problem dimensions. We will also see experimental results comparing optimization time on the quantum computer against optimization time on a classical computer. Those are the first experimental results on a quantum annealer that were ever published for an optimization problem from the database domain.

## 8.1 Introduction

The database area has motivated a multitude of hard optimization problems that probably cannot be solved in polynomial time. Those optimization problems become harder as data processing systems become more complex. This makes it interesting to explore also unconventional optimization approaches. In this chapter, we explore the potential of quantum computing for a classical database-related optimization problem, the problem of multiple query optimization (MQO) [119]. We were granted a limited amount of computation time on a D-Wave 2X adiabatic quantum annealer, currently hosted at NASA Ames Research Center in California. This device is claimed to exploit the laws of quantum physics [29] in the hope to solve NP-hard optimization problems faster than traditional approaches. The machine supports a very restrictive class of optimization problems while it is for instance not capable of running Shor's algorithm [123] for factoring large numbers[1]. We will show how instances of the multiple query optimization problem can be brought into a representation that is suitable as input to the quantum annealer. We also report results of an experimental evaluation that compares the time it takes to solve MQO problems on the quantum annealer to the time taken by algorithms that run on a traditional computer. We believe that this is the first experimental evaluation on a quantum computer for an optimization problem in the database community.

The quantum annealer, produced by the Canadian company D-Wave[2], uses *qubits* instead of bits. While bits have a deterministic value (either 0 or 1) at each point in time during a computation, a qubit may be put into a superposition of states (0 *and* 1) that would be considered mutually exclusive according to the laws of classical physics. Working with qubits instead of bits could in principle allow faster optimization than on a classical computer [9]. Thinking of qubit superposition as a specific form of parallelization is certainly simplifying but still gives a first intuition for why this is possible. We provide more explanations on quantum computing and on the quantum annealer in Section 8.2.

The quantum annealer that we were experimenting with has a net worth of around 15 million US dollars. This price might make main stream adoption seem illusory in the near-term future. However, the company D-Wave is currently considering flexible provisioning models allowing users to buy computation time instead of the hardware[3]. In this scenario, users would use the machine remotely, in a similar way as we did in our experiments. As near-optimal solutions to hard problems can usually be found within milliseconds (see Section 8.7), this provisioning model might allow optimization at an affordable rate per instance. Those are some of the factors that encourage us to explore the potential of quantum computing already at this point in time.

The D-Wave adiabatic quantum annealer has been the subject of controversial discussions in the scientific community. Those discussions have focused on two questions: whether quantum effects play indeed a significant role during the optimization process [14, 29, 30, 93, 122, 128] and whether the performance is significantly better than the one of classical computers [71, 86, 85]. Recent publications seem to answer the first question positively [14, 30, 50, 93], as acknowledged by MIT professor and D-Wave critic Scott Aaronson ("this completely nails down the case for computationally-relevant collective quantum tunneling in the D-Wave machine"[4]) and other experts. The answer to the second question depends apparently on the specific class of problems considered, leading for instance to different conclusions for range-limited Ising problems [85] than for Ising problems without weight limits [71]. Solving problem classes that are not natively supported by the quantum annealer requires transformation steps which add a problem class-specific overhead in the problem representation size that might prevent the quantum annealer from solving non-trivial instances due to its limited number

[1]http://www.dwavesys.com/blog/2014/11/response-worlds-first-quantum-computer-buyers-guide
[2]http://www.dwavesys.com/
[3]http://spectrum.ieee.org/podcast/computing/hardware/dwave-aims-to-bring-quantum-computing-to-the-cloud
[4]http://www.scottaaronson.com/blog/?p=2555

```
1: // Solves multiple query optimization problem $M$
2: function QuantumMQO($M$)
3:     // Map MQO problem to logical energy formula
4:     $lef$ ←LogicalMapping($M$)
5:     // Map logical into physical energy formula
6:     $pef$ ←PhysicalMapping($lef$)
7:     // Minimize formula on quantum computer
8:     $b_i$ ←QuantumAnnealing($pef$)
9:     // Transform physical into logical solution
10:    $X_p$ ←PhysicalMapping$^{-1}$($b_i$)
11:    // Transform logical solution to MQO solution
12:    $P_e$ ←LogicalMapping$^{-1}$($X_p$)
13:    // Return best set of query plans to execute
14:    return $P_e$
15: end function
```

Algorithm 19 – How to solve multiple query optimization on an adiabatic quantum annealer.

of qubits. Our work adds to the discussion concerning the second question by providing a mapping algorithm and experimental results for a specific database-related optimization problem.

Prior work on MQO [19, 44, 46, 54, 53, 53, 83, 95, 99, 114, 119, 121] did not consider the potential of quantum computing. Prior publications in the area of quantum computing [25, 58, 63, 96, 111, 143, 127] did not treat the MQO problem.

Algorithm 19 shows the high-level approach by which we obtain solutions to MQO problem instances from a quantum annealer. The goal of MQO is to select the optimal combination of query plans to execute in order to minimize execution cost for a batch of queries. Given an MQO problem instance $M$, we introduce binary variables $X_p$ for each available query plan $p$ that indicate whether the corresponding plan is executed. We transform the given MQO instance into an energy formula (the term derives from the fact that the quantum annealer translates such formulas into energy levels) on those variables that becomes minimal for a variable assignment representing an optimal solution to the initial MQO problem $M$. We call the variables $X_p$ the logical variables to indicate that they cannot yet be represented by single qubits within the qubit matrix of the quantum annealer. We call the transformation the *logical mapping* and the resulting formula the *logical energy formula*.

The physical mapping transforms the logical energy formula, defined in the variables $X_p$, into a *physical energy formula* that uses the binary variables $b_i$. Each variable $b_i$ is associated with one specific, physical qubit of the quantum annealer. Finding a value assignment for the variables $b_i$ which minimizes the physical energy formula is an NP-hard problem. We use the quantum annealer to solve it. All other transformations depicted in Algorithm 19 have polynomial complexity and are executed on a classical computer.

Based on the solution returned by the quantum annealer, the value assignment to the variables $b_i$ which minimizes the physical energy formula, we transform the solution to the physical energy formula into a solution to the logical energy formula. Finally, we transform the solution to the logical energy formula into a solution to the original MQO problem which is the optimal set $P_e$ of query plans to execute.

MQO problems cannot be solved with our approach if the number of qubits required by the physical energy formula exceeds the number of qubits available on the quantum annealer. Albeit doubling the number of qubits compared to the predecessor model, the number of qubits is with slightly over one thousand qubits still very limited on the D-Wave 2X that we experimented with. Correspondingly, the limited number of qubits is in practice the most important factor restricting the size of the problem instances that can be treated with the quantum annealer. For that reason, we analyze the "complexity" of our mapping algorithm in terms of the asymptotic growth rate of the number of required qubits as a function of the MQO problem dimensions. This approach is common in the area of quantum annealing [96, 127]. We find that the number of qubits in the physical energy formula grows quadratically in the number of plans per query and at least linearly in the number of queries.

In our experimental evaluation, we compare our approach based on quantum annealing against classical optimization algorithms executed on traditional computers. We compare against classes of algorithms that have been proposed for MQO in prior publications and include integer linear programming, genetic algorithms, and simple greedy heuristics. While the number of available qubits severely limits the class of non-trivial MQO problems that can be treated efficiently on the quantum annealer, we also find a class of problems where the quantum annealer discovers near-optimal solutions at least 1000 times faster than classical approaches.

In summary, our original scientific contributions in this chapter are the following:

- We map MQO problem instances into a representation that can be solved on a quantum annealer.
- We analyze the complexity of our mapping method in terms of the asymptotic number of required qubits as a function of the MQO problem dimensions.
- We experimentally compare the D-Wave 2X quantum annealer against competing approaches for MQO.

The remainder of this chapter is organized as follows. In Section 8.2, we give a short introduction to quantum computing and the quantum annealer. In Section 8.3, we introduce our formal problem model for MQO. We describe the logical mapping in Section 8.4 and the physical mapping in Section 8.5. We formally prove correctness of our mapping and analyze the asymptotic complexity in Section 8.6. In Section 8.7, we evaluate our approach experimentally. We discuss related work in Section 8.8 and conclude in Section 8.9.

## 8.2 Quantum Computing

We give a short introduction to quantum computing in general and to the specific realization inside the D-Wave quantum annealer. Our goal is to provide the reader with a rough intuition while we simplify many of the details. A detailed introduction to those complex topics is beyond the scope of this dissertation and we refer interested readers to specialized publications [9].

Quantum mechanics describes physical processes at extremely small scale. The laws of quantum mechanics do not match our intuition since our intuition is formed by the macroscopic world. For instance, extremely small particles may at the same time adopt two states that are mutually exclusive according to our normal intuition.

Quantum computers [9] are machines that harness quantum physics to potentially achieve speedups over classical computers. Classical computers use bits that are in either one of two states (1 or 0); quantum computers use qubits that can at the same time be set to 1 *and* to 0, a state that we call *superposition*. This allows quantum computers to explore many alternative computational branches at the same time and there are problems (e.g., prime factorization [123]) for which quantum algorithms provide an exponential speedup over the best currently known classical algorithms.

The first commercially available machine claimed to harness quantum effects to speed up optimization is the quantum annealer by D-Wave Systems. In order to use the D-Wave quantum annealer, each optimization problem must be represented as a mathematical function with binary variables. The D-Wave computer aims to find the variable value assignments minimizing the given function.

More precisely, the D-Wave computer minimizes sums of terms that are either linear or quadratic in the output variables. This problem model corresponds to the quadratic unconstrained binary optimization problem which is NP-hard. The following explanations of the internal workings of the D-Wave machine show that this choice of input format is intrinsically imposed by the D-Wave architecture.

The D-Wave machine represents binary variables as qubits. Qubits are realized as tiny electric circuits. Those circuits are cooled down to a temperature of 13 millikelvin. Quantum effects appear at this temperature and the current may flow at the same time clockwise and counterclockwise within the circuits, thereby representing qubit superposition. The input function that needs to be minimized is translated into magnetic fields affecting single qubits or qubit pairs. Fields affecting single qubits represent linear terms while fields affecting qubit pairs represent quadratic terms. The strength of those magnetic fields is tuned to be proportional to the weights assigned to the corresponding terms in the input function. Thereby we obtain a physical system that minimizes its total energy for qubit states that represent variable assignments minimizing the input function.

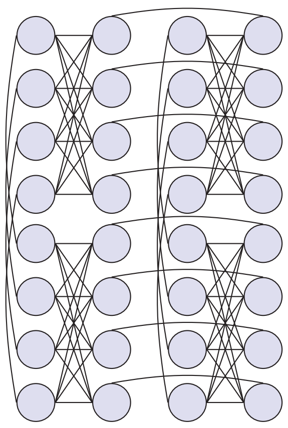


Figure 8.1 – Four neighboring unit cells containing eight qubits each, connected in a Chimera structure.

The goal of minimizing the input function is translated into the goal of minimizing the energy level within a physical system in which quantum effects are present. In order to reach the minimal energy level (and thereby solving the input problem), the D-Wave computer executes a process called quantum annealing.

We introduce quantum annealing informally by contrasting it from the simulated annealing algorithm (SA) which is a classic heuristic optimization algorithm. SA simulates thermal annealing in software while D-Wave performs actual quantum annealing in hardware. Both annealing algorithms process an energy function with the goal to find its global minimum. The SA algorithm performs a set of moves in the search space, using evaluations of the given cost function as guidance in the hope to eventually reach a global minimum. The quantum annealing algorithm starts instead with a simplified cost function whose global minimum can be easily calculated. During optimization, the quantum annealing algorithm does not perform moves in the search space but rather transforms the cost function slowly from the initialization function to the cost function of interest. During that process, the quantum annealer is in a superposition of possible states, unlike its deterministic counterpart. If this transformation is executed slowly enough and without disturbances then the quantum annealer is guaranteed to remain within the global minimum throughout the whole transformation [56] which can be read out after annealing terminates. In practice, annealing runs are often disturbed despite all shielding efforts and a multitude of runs must be executed before finding an optimal solution.

We executed our experiments on the D-Wave 2X which was released in August 2015 and is the most recent model in a series of quantum annealers presented by D-Wave with a net price of around 15 million dollars. The number of qubits has been roughly doubling from one model to the next over the past years and the D-Wave 2X features a matrix of 1152 interconnected qubits. The manufacturing process is currently imperfect and only 1097 out of 1152 qubits were fully functional on the machine that we used. Connections between qubits are sparse and form the so called Chimera graph [98]. Figure 8.1 shows an extract of the Chimera graph structure as it is available on the qubit matrix of the quantum annealer. Qubits are partitioned into so called unit cells. Each unit cell contains eight qubits in two colons and connects each qubit to all four qubits in the opposite colon but not to qubits in the same colon. Qubits in the left colon are connected to their respective counterpart in the qubit cell above and below while qubits in the right colon are connected to their counterparts in the cells to the right and to the left (unless it is the border of the qubit matrix). Each qubit is hence connected to at most six other qubits. The D-Wave 2X uses 144 unit cells.

## 8.3 Formal Model

The goal in multiple query optimization (MQO) is to minimize the joint execution cost for a batch of queries by exploiting possibilities to share computation between different queries [117]. Our MQO problem model is based on standard assumption [117]: we assume that a small set of alternative plans has been found for each query prior to MQO and that execution costs of query plans can be reliably estimated.

An MQO problem instance is characterized by a set $Q$ of queries. Each query either represents a final result that is requested by the user or an intermediate result that is useful when generating final results. Each query $q \in Q$ is associated with a set of alternative generation plans $P_q$. For a final result, all associated plans must represent methods of generating that result. The generation of intermediate results is optional and the plan set of an intermediate result may contain one plan that represents the possibility of not generating that result.

Set $P = \cup_q P_q$ denotes the set of all considered plans. Each plan $p \in P$ is associated with an execution cost $c_p$. This is the cost of processing the plan without exploiting any previously generated intermediate results. Plans for different queries may however share partial results. It is beneficial to select groups of plans that can share many intermediate results to reduce processing cost. Given two plans $p_1$ and $p_2$ that can share intermediate results, we denote by $s_{p1,p2} > 0$ the cost reduction that can be achieved by sharing. Note that our model is not restricted to the case that two plans share an intermediate result. If more than two plans can share an intermediate result, we introduce pair-wise connections between the result and all plans that may share it.

A solution to an MQO problem instance is a subset of plans $P_e \subseteq P$ that are selected for execution. A solution is only valid if exactly one plan is selected for each query and $\forall q \in Q : |P_q \cap P_e| = 1$. As discussed before, selecting a plan does not necessarily mean that the corresponding result is generated in case of intermediate results. We denote the accumulated execution cost of a plan set by $C(P_e) = \sum_{p \in P_e} c_p - \sum_{\{p1,p2\} \subseteq P_e} s_{p1,p2}$. A solution is optimal if its execution cost is minimal among all valid solutions.

We selected an MQO problem model that shortens our following descriptions of the transformation into a quadratic unconstrained binary optimization (QUBO) problem, the formalism that we introduce next. Our MQO model is however equivalent to MQO problem models that were used in prior work[5] and the problem remains NP-hard.

A QUBO problem is defined over a set $\{X_i\}$ of binary variables (with value domain $\{0,1\}$). A solution to a QUBO problem assigns each of the variables to one of the two possible values. The goal is to minimize the following function that depends on the binary variables: $\sum_{i\leq j} w_{ij} X_i X_j$. The weights $w_{ij}$ are problem instance specific. Note that the formula contains linear terms (for $i = j$ since $x_i^2 = x_i$ for binary variables) as well as quadratic terms (for $i \neq j$). A solution to a QUBO problem is optimal if it minimizes the function from above among all possible solutions.

## 8.4 Logical Mapping

We show now how to transform an MQO problem instance into a QUBO problem instance. This step is required since the quantum annealer can only solve QUBO problems.

As discussed in Section 8.3, an MQO problem is defined by a set $Q$ of queries, a set $P_q$ of plans for each query $q \in Q$ with $P = \cup_q P_q$, execution cost values $c_p$ for each plan $p \in P$, and possible cost savings $s_{p1,p2}$ for each plan pair $p1, p2$. A solution is a subset of plans that are selected for execution such that one plan is selected per query.

Only binary variables may appear in a QUBO problem. We must therefore represent the solution space of the MQO problem using binary variables. Given a set $P$ of plans, we introduce a binary variable $X_p$ for each plan $p \in P$. If $X_p = 1$ then plan $p$ is selected for execution while $p$ is not executed if $X_p = 0$.

An MQO solution is only valid if exactly one plan is selected for each query. If our goal was to transform an MQO problem into an integer linear program, we could introduce constraints of the form $\sum_{p\in P_q} X_p = 1$ for each $q \in Q$ to guarantee that all returned solutions are valid. Unfortunately, the QUBO formalism does not allow to express constraints directly. As the optimal solution to a QUBO problem minimizes a quadratic formula, we can however add terms to that formula that take high values if constraints are violated. This approach guarantees a valid solution if those terms are scaled up by sufficiently high weights.

We call the quadratic formula defining the QUBO problem short the *energy formula* in the following as it is translated into energy levels by the D-Wave annealer. We decompose the constraint that exactly one plan is selected per query into two parts: we require that at least one plan is selected and that at most one plan is selected. In order to assure that at least one plan is selected for each query, we can simply add the term $E_L = -\sum_{p\in P} X_p$ to the energy formula. As lower values of the energy formula are preferable, this term motivates to set all variables $X_p$ to one. We can express the constraint that at most one plan is selected by adding

[5] If each query plan is modeled by a set of tasks [119] then we make in our model the execution cost of the plan equal to the sum of the execution costs of all tasks and introduce one extra query for each of the tasks with an execution cost equal to the task cost and a cost savings link between task and plan whose value equals the task execution cost again.

the term $E_M = \sum_{q\in Q}\sum_{\{p1,p2\}\subseteq P_q} X_{p1}X_{p2}$ to the energy formula. This term takes value zero if at most one plan is selected per query and at least value one otherwise. As we will discuss in the following paragraphs, both terms will have to be scaled by an appropriate factor to make sure that all constraints are respected.

The terms that we have so far inserted into the energy formula make sure that a valid solution is preferable compared to an invalid solution. The goal of the MQO problem is however to minimize execution cost. We must introduce additional energy terms to make a valid solution with lower execution cost preferable over a valid solution with higher execution cost.

We take into account plan execution cost by introducing the term $E_C = \sum_{p\in P} c_p X_p$ into the energy formula. This means that the execution cost of each selected plan $p$ with $X_p = 1$ is added. On the other hand, we must introduce the term $E_S = -\sum_{\{p1,p2\}\subseteq P} s_{p1,p2}X_{p1}X_{p2}$ to represent the possibility of sharing intermediate results between plans. We finally scale up the first two terms that we introduced by a factor whose value we discuss in the following. The resulting energy formula reads

$$w_L E_L + w_M E_M + E_C + E_S.$$

We discuss in the following how to choose the weights $w_L$ and $w_M$. It is crucial to choose the weights as low as possible since having high weights seems to increase the chances of obtaining sub-optimal solutions from the quantum annealer [85]. We will derive inequalities of the form $w > a$ in the following where $w$ is a weight and $a$ a value that lower-bounds the admissible weights. Having such an inequality, we prefer for the aforementioned reason to choose $w = a + \varepsilon$ in general where $\varepsilon$ is a small value (we typically use $\varepsilon = 0.25$ in our implementation).

The energy formula contains two terms that motivate valid solutions ($E_L$ and $E_M$) and two terms that motivate solutions with lower execution cost ($E_C$ and $E_S$). The terms motivating a valid solution should intuitively obtain higher weights than the ones motivating low-cost solutions. If the terms enforcing valid solutions are not associated with sufficiently high weights then the optimal QUBO solution might not select any plans to save execution cost.

We must make sure that the motivation of selecting at least one plan is always higher than the motivation to save execution cost by not selecting any plan. We accomplish this by requiring $w_L > \max_{p\in P} c_p$. Having scaled up $E_L$ by that factor, the partial energy formula $w_L E_L + E_C + E_S$ would be minimized by executing each plan for each query. This does clearly not reflect the original MQO problem and we must add $w_M E_M$ to restrict the number of plan selections per query to one.

Clearly we must choose $w_M > w_L$ to accomplish the aforementioned goal. This is however insufficient. The generation cost of a query can be lower than the cost reduction achievable by sharing it among other plans. Hence, even if we have $w_M > w_L$, the energy formula might still be minimized by executing multiple plans for the same query. This is due to a shortcoming of the QUBO representation: the QUBO representation leads to believe that it is possible to accumulate cost savings by generating the same result according to multiple plans. In reality, this is of course not the case. We circumvent that problem by explicitly enforcing that at most one plan is selected per query. This is guaranteed if $c_M > c_L + \max_{p1\in P}\sum_{p2\in P} s_{p1,p2}$.

**Example 17.** *We show how to transform a simple MQO problem into the QUBO representation. Assume that four plans $p_1$, $p_2$, $p_3$, and $p_4$ are considered with execution cost 2, 4, 3, 1 respectively. The first two plans generate query $q_1$ and the next two plans generate query $q_2$. Assume further that $p_2$ and $p_3$ can share an intermediate result allowing cost savings of 5 cost units. The QUBO representation uses the binary variables $X_1$, $X_2$, $X_3$, and $X_4$ that are associated with plans $p_1$ to $p_4$ and are set to one if the corresponding plan is executed. Then execution cost is represented by the term $E_C = 2X_1 + 4X_2 + 3X_3 + 1X_4$. Potential cost savings are represented by the term $E_S = -5X_2X_3$. The term $E_L = -\sum_{i=1}^{4} X_i$ enforces at least one plan selection for each of the two queries and is weighted by factor $w_L = 4 + \varepsilon$. Term $E_M = X_1X_2 + X_3X_4$ enforces at most one plan selection if weighted by factor $w_M = w_L + 5$. The variable assignment $X_1 = 0$, $X_2 = 1$, $X_3 = 1$, $X_4 = 0$ minimizes the energy formula and represents the optimal solution to the MQO problem at the same time.*

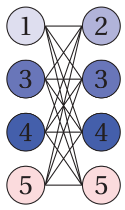
We prove formally in Section 8.6 that the mapping method presented in this section is correct.

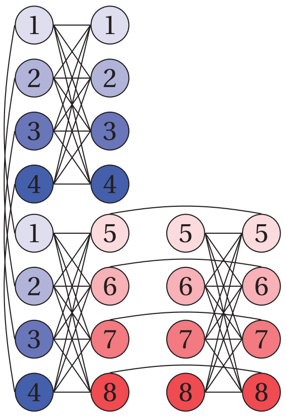
## 8.5 Physical Mapping

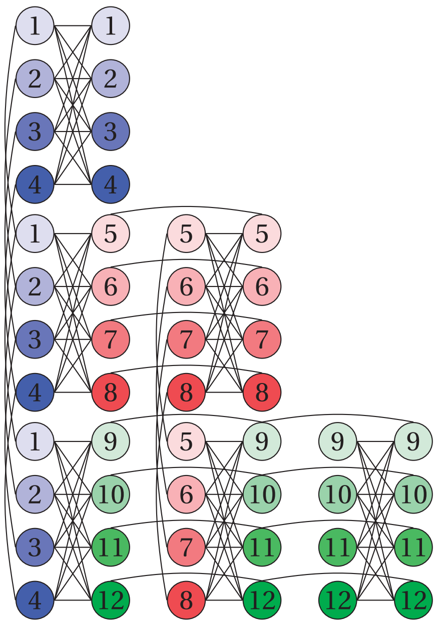
We have seen in the previous section how to transform an MQO problem into an energy formula defined on the variables $X_p$. We require one more transformation until we can apply the quantum annealer: we must choose for each logical variable $X_p$ a group of physical qubits to represent it. Then we must set the weights on single qubits and the strengths of the couplings between qubits in order to translate the *logical energy formula* of the form $\sum_{\{p1,p2\}\subseteq P} w_{p1,p2}X_{p1}X_{p2}$ into a *physical energy formula* of the form $\sum_{i\leq j}\widetilde{w}_{ij}b_ib_j$ where $b_i$ represents the state of the $i$-th qubit of the quantum annealer. We call this transformation the *physical mapping* or *embedding*.

This second transformation is required since it is in general insufficient to represent one QUBO variable by one qubit. This is due to the sparse connection structure between qubits (see Section 8.2 for a detailed description of the connection structure). Each qubit is connected to at most six other qubits. In the physical energy formula, only the weights between connected qubits can be different from zero. Hence for a fixed $i$ there are at most six values for $j$ such that $\widetilde{w}_{ij} \neq 0$. If a QUBO variable interacts with more than six other QUBO variables (meaning that for a fixed plan $p1$ there are more than six plans $p2$ such that $w_{p1,p2} \neq 0$ in the logical energy formula) then we must represent that variable by multiple qubits.

The physical mapping consists of three steps. First, for each QUBO variable we must select a group of physical qubits to represent it. Second, if the energy formula contains a term of the


(a) TRIAD with 5 chains


(b) TRIAD with 8 chains


(c) TRIAD with 12 chains

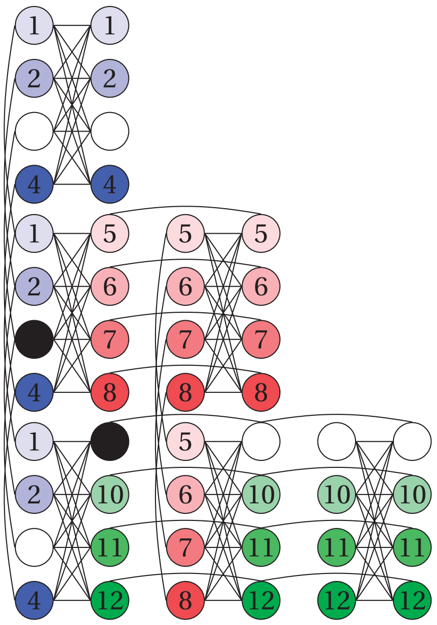

(d) TRIAD with 12 chains and two broken qubits

Figure 8.2 – TRIAD pattern in different sizes: we show qubits as circles, annotated by the ID of the logical variable that they represent. The mapping from variables to qubits assures that each variable shares at least one connection (in black) with each of the other variables.

form $w_i X_i$ where $w_i$ is a weight and $X_i$ a QUBO variable then we must distribute that weight over all qubits representing $X_i$: if $B$ denotes the set of qubits representing $X_i$ then the weight $w_i/|B|$ is added on each qubit in $B$. If a term $w_{ij}X_iX_j$ appears in the energy formula then we select one qubit $b_1$ among the qubits representing $X_i$ and another qubit $b_2$ among the qubits representing $X_j$ such that $b_1$ and $b_2$ are connected by a coupling in the qubit matrix and we increase the strength of that coupling by $w_{ij}$. In a third step, we must make sure that all qubits representing the same variable "behave as one bit" and are assigned consistently to the same value after an annealing run. We accomplish this by adding additional weights on the qubits and on the couplings between qubits representing the same variable such that the minimum energy is reached for a consistent assignment. This requires that all qubits representing the same variable form a chain of connected qubits.

We provide further details on step one and step three in the following, starting with step one.

Mapping variables to qubits is a highly non-trivial problem as the mapping must satisfy various constraints. First, we must represent variables by groups of qubits that are connected in a chain. Second, if two logical variables appear together in a quadratic term in the energy formula then the two groups of qubits representing those variables must be connected, i.e. at least one qubit from the first group is connected to at least one qubit of the second group. Third, we must take into account that some of the qubits and inter-qubit connections on the D-Wave annealer are broken and cannot be used (see Figure 8.1).

Finding for a given QUBO problem the embedding that satisfies all of the aforementioned constraints while consuming the minimal number of qubits is an NP-hard problem [89]. We cannot solve it optimally without the risk that the time for finding an optimal embedding dominates the time of finding the optimal solution to the resulting QUBO problem. For that reason, we currently use simple embedding schemes that can be generated with negligible time overhead and are presented in the following.

Figure 8.2 shows the TRIAD pattern proposed by Choi [39] in the graphical representation introduced by Venturelli et al. [142]. This pattern allows to embed arbitrary QUBO problems. Figures 8.2a to 8.2c show the pattern in different sizes, supporting 5, 8, and 12 logical variables. When representing each logical variable by a chain of qubits in this pattern (qubits in the same chain are labeled by the same number in the figure) then arbitrary energy formulas can be modeled since the pattern connects each pair of variables.

The method currently used for manufacturing the qubit matrix is imperfect and results in a certain percentage of broken qubits. If a qubit chain contains broken qubits then the entire chain becomes unusable since it cannot be guaranteed anymore that all qubits in the chain are assigned to the same value. Figure 8.2d illustrates the problem, visualizing broken qubits in black and intact qubits in unusable chains in white.

All chains in the TRIAD pattern are connected by at least one coupling. The downside of enabling so many connections is that the number of qubits consumed by the TRIAD pattern grows quadratically in the number of chains and qubits must be considered a scarce resource

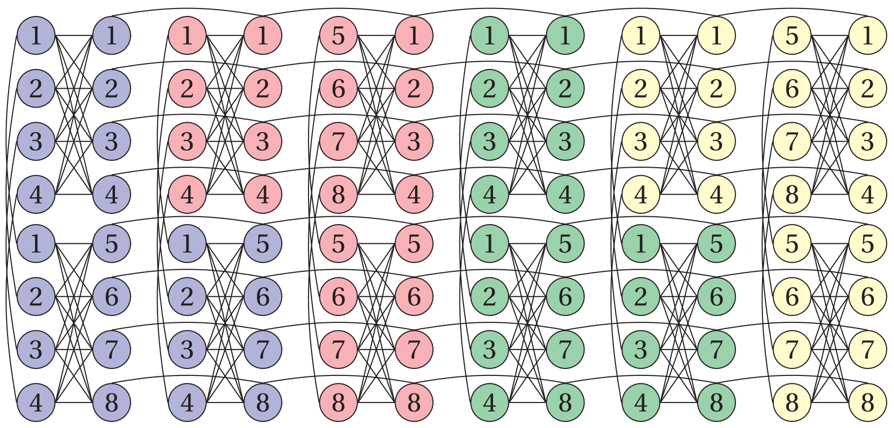


Figure 8.3 – Clustered embedding pattern: qubits representing plans in different clusters are distinguished by their color (four colors hence four clusters), the qubit label is the plan identifier (numbers one to eight represent eight alternative plans per cluster).

on current quantum annealers. Analyzing the energy formula from the last section, we find that we require connections between logical variables representing different plans for the same query (due do the quadratic sub-terms contained in $E_M$) and connections between variables representing plans for different queries with work overlap (due to the terms in $E_S$).

Existing approaches for MQO cluster queries based on structural properties in a preprocessing step [95] such that queries in different clusters are less likely to share intermediate results. We can exploit such a clustering in certain cases as illustrated in Figure 8.3: instead of a single TRIAD pattern, we use multiple TRIAD patterns where each TRIAD represents all variables associated with the plans for the query in one single cluster. As different plans for the same query are integrated into the same TRIAD structure, we are sure to realize all connections required by term $E_M$. As plans for different queries in the same cluster are integrated into the same TRIAD as well, all connections required by term $E_S$ can be realized, too. The connections between qubits representing plans in different clusters are sparse but so are the opportunities of work sharing between them and connections between plans in different clusters can only represent work sharing opportunities. The advantage of using the clustered pattern over the single TRIAD pattern is that the number of required qubits grows more slowly in the number of queries and plans as we analyze in more detail in the following section.

The annealing process that is executed by the quantum annealer takes only into account the physical energy formula. We cannot directly integrate the information that multiple qubits represent the same logical variable and should be assigned to the same value. Instead, we must add more terms to the physical energy formula that make groups of qubits “behave as one bit”. More precisely, we add, for each group of qubits, terms to the energy formula that take high values if the qubits are assigned to different values. As the goal is to minimize the energy formula, such terms will drive the annealing process towards solutions that assign groups of qubits representing the same variable to the same value. Then we can read out one single value for the entire qubit group and associate it with the represented variable.

Assume that two connected qubits $b_1$ and $b_2$ represent the same variable. We motivate assigning the same value to both of them by adding the term $b_1 + b_2 - 2b_1b_2$ to the energy formula. This term takes value one if the qubits are assigned to different values and takes value zero if both qubits are assigned to the same value.

Assume now that we have a group of qubits with more than two elements that need to be assigned to the same value. We generally require qubit groups representing the same variable to form a chain. This means that we can order the qubits into a sequence $\langle b_1, b_2, \ldots, b_m\rangle$ such that each qubit $b_i$ is connected to its successor $b_{i+1}$ in the qubit matrix. Under this assumption, we can add energy terms of the form $E_B(i) = b_i + b_{i+1} - 2b_ib_{i+1}$ that motivate assigning the same value to two consecutive qubits. Adding the terms $E_B = \sum_{i=1}^{m-1} E_B(i)$ to the energy formula motivates assigning all qubits to the same value.

We assume in the following that the terms from the logical energy formula have already been integrated into the physical energy formula as described under step two at the beginning of this section. Hence the physical energy formula contains terms in addition to the terms $E_B$. This means that we have to scale up the terms $E_B$ by a factor that is sufficiently high to assure that the energy formula becomes minimal for a value assignment where all equality constraints, represented by $E_B$, are satisfied. As discussed in Section 8.4, we choose the scaling factors as low as possible to avoid a large range of possible energy values.

The following scaling method is based on ideas by Choi [38]. We treat each group $B$ of qubits representing the same variable separately and calculate a specific scaling factor for $E_B$. This scaling factor must make sure that a solution with inconsistent assignments for a qubit group improves (i.e., the value of the energy formula decreases) once replacing inconsistent assignments by a consistent one (either all qubits in the group are set to one or all are set to zero). Consider a group $B$ of qubits representing the same variable that are assigned to inconsistent values. The term $w_BE_B$ adds at least $w_B$ to the energy formula as the chain must be broken at least at one position. Replacing the inconsistent assignment by a consistent assignment lets $w_BE_B$ take the value zero and reduces the total energy level by $w_B$.

Making the assignment for $B$ consistent must reduce the energy value of $w_BE_B$ but it might increase the value of other terms in the energy formula. We calculate in the following the upper bound $U$ on the increase in the other energy terms. We denote by $U_{0\to 1}(b)$ the maximal increase in energy caused by changing the value of $b$ from zero to one. Denote by $v$ the weight on $b$ and by $v_i$ all weights on couplings that connect $b$ to qubits outside of $B$. Then we have $U_{0\to 1}(b) = v + \sum_i \max(v_i, 0)$. We pessimistically assume here that each qubit connected to $b$ via a positive weight is set to one while qubits connected via a negative weight are set to zero. This yields an upper bound on the increase in energy. We can calculate an upper bound for the energy increase when setting the value of $b$ from one to zero in the analogue fashion and denote the result by $U_{1\to 0}(b)$.

We have the choice between setting all qubits in $B$ to one or setting all of them to zero in order to make the assignment for $B$ consistent. We can select the option that leads to a lower increase in energy and therefore obtain $U = \min(\sum_{b\in B} U_{1\rightarrow 0}(b), \sum_{b\in B} U_{0\rightarrow 1}(b))$ as upper bound for the increase in all energy terms except for $E_B$ when making an inconsistent assignment consistent. This means that the energy formula must become minimal for a consistent assignment if we set $w_B = U + \varepsilon$.

## 8.6 Formal Analysis

In this section, we prove that the transformation from MQO to QUBO problems that we introduced in Section 8.4 is correct, meaning that the optimal solution to the QUBO problem represents indeed the optimal solution to the MQO problem. Later, we will analyze how the number of qubits that our mapping requires evolves as a function of the MQO problem dimensions.

We prove that the energy formula $w_M E_M + w_L E_L + E_C + E_S$ becomes minimal for an assignment of variables to values representing an optimal solution to the MQO problem from which the energy formula was derived.

**Lemma 18.** *The energy formula is minimized by selecting at most one plan per query.*

*Proof.* Assume that the energy formula was minimized by setting $X_{p1} = X_{p2} = 1$ where $p1$ and $p2$ are alternative plans for the same query. If we set $X_{p1} = 0$ (or $X_{p2}$) then the value of term $E_C$ decreases by the execution cost of $p1$ while $E_S$ might increase as cost savings enabled by executing $p1$ cannot be realized. Term $E_S$ increases at most by the accumulated cost savings enabled by $p1$ which is $\sum_{p\in P} s_{p1,p}$. The value of term $w_L E_L$ increases by $w_L$. Term $E_M$ contains the sub-term $w_M X_{p1} X_{p2}$ so the value of $E_M$ decreases by $w_M$. In summary, the energy increases at most by $w_L + \sum_{p\in P} s_{p1,p}$ while it decreases by $w_M$ and $w_M > w_L + \sum_{p\in P} s_{p1,p}$. The energy decreases by setting $X_{p1} = 0$ which contradicts our initial assumption. □

**Lemma 19.** *The energy formula is minimized by selecting at least one plan per query.*

*Proof.* Assume that the energy formula was minimized by setting $X_p = 0$ for all $p \in P_q$ for a query $q \in Q$. Pick one arbitrary plan $p \in P_q$ and set $X_p = 1$ instead. Then the value of term $E_C$ increases by the execution cost $c_p$ of that plan. The value of term $E_S$ can only decrease since executing $p$ might enable possibilities to share work and reduce execution cost. The value of $E_M$ remains constant while the value of $w_L E_L$ decreases by $w_L$. In summary, the energy increases at most by $c_p$ and decreases by $w_L$ and $w_L > c_p$. The energy decreases by setting $X_p = 1$ which contradicts our initial assumption. □

**Theorem 37.** *The energy formula is minimized for a valid solution with minimal execution cost.*

*Proof.* The energy formula becomes minimal for a valid solution (meaning that one plan is selected per query) according to Lemmata 18 and 19. Furthermore, terms $E_L$ and $E_M$ have the same value for each valid solution (value $-|Q|$ for $E_L$ and value 0 for $E_M$). This means that those two terms do not influence the choice between valid solutions. The selection of an optimal solution is entirely governed by the combined term $E_C + E_S$. The theorem follows since that term represents exactly the execution cost, taking into account cost reductions by shared work. □

It is common to analyze the time complexity of optimization algorithms on traditional (non-quantum) computers. It would be interesting to analyze the asymptotic run time until the quantum annealer finds optimal or near-optimal solutions as a function of the MQO problem dimensions. A theoretical framework for analyzing worst case time complexity on adiabatic quantum computers is however currently not available [143]. This is why the analysis of adiabatic quantum approaches usually focuses on the number of required qubits. This metric is relevant since the number of available qubits imposes most restrictions in practice.

We analyze the required number of qubits as a function of the following variables. We denote by $n$ the number of query clusters, by $m$ the number of queries per cluster, and by $l$ the number of alternative plans per query. We generally assume that connections between plans in the same cluster are relatively dense while connections between different clusters are relatively sparse. In order to simplify the following analysis, we assume now the extreme case that all plans in each cluster are connected while no connections exist between different clusters. This would allow to decompose the problem and treat different clusters separately but the following results still apply to sparsely connected clusters in which decomposition is not possible. We first analyze the QUBO representation from Section 8.4 in terms of how many qubits it *minimally* requires.

**Theorem 38.** *The QUBO representation introduced in Section 8.4 requires* $\Omega(n \cdot (m \cdot l)^2)$ *qubits.*

*Proof.* The total number of considered plans is $n \cdot m \cdot l$. This is at the same time the number of logical variables and hence a lower bound on the number of qubits (as each variable must be represented at least by one qubit). We must however take into account that each qubit is connected to at most six other qubits. The number of required connections between logical variables leads therefore to another lower bound on the required number of qubits.

Only quadratic terms in the energy formula require connected qubits. Terms $E_L$ and $E_C$ contain no quadratic sub-terms while $E_M$ connects all plans for the same query and term $E_S$ connects plans with work overlap. As mentioned before, we simplify by assuming that $E_S$ connects all plans in the same cluster but no plans in different clusters. Hence plans are connected to all plans in the same cluster. The number of plans per cluster is $m \cdot l$ so each plan is connected to $\Omega(m \cdot l)$ other plans. Due to the constant number of connections per qubit, this means that each plan must be represented by $\Omega(m \cdot l)$ qubits. Multiplying by the total number of plans, $n \cdot m \cdot l$, yields the postulated result. □

The minimal number of qubits is a property of the logical mapping presented in Section 8.4. Now we analyze the actual (asymptotic) number of qubits required by the clustered mapping pattern presented in Section 8.5. We assume that all qubits used by the pattern are intact.

**Theorem 39.** *The physical mapping pattern introduced in Section 8.5 requires* $\Theta(n \cdot (m \cdot l)^2)$ *qubits.*

*Sketch.* The plans in each cluster are mapped to a TRIAD pattern. We can prove by induction that the number of qubits required by a TRIAD grows quadratically in the number of chains. The number of plans per cluster is $m \cdot l$ so the number of qubits per cluster is in $\Theta((m \cdot l)^2)$. Multiplying by the number of clusters yields the result. □

We see that the asymptotic number of qubits required by our physical mapping matches the lower bound. We finally analyze the time complexity of the preprocessing phase that is executed on a classical computer.

**Theorem 40.** *Calculating the physical energy formula is in* $O(n \cdot (m \cdot l)^2)$ *time.*

*Proof.* We first analyze the time complexity of the logical mapping. The energy formula consists of the terms $E_L$, $E_M$, $E_C$, and $E_S$. The time complexity for calculating the weights for those terms is proportional to the number of linear and quadratic sub-terms plus the complexity of calculating scaling factors. The terms $E_L$ and $E_C$ contain $n \cdot m \cdot l$ sub-terms respectively, term $E_M$ contains $O(n \cdot m \cdot l^2)$ sub-terms, and term $E_S$ contains $O(n \cdot (m \cdot l)^2)$ sub-terms. Calculating $w_L$ requires to determine the maximum out of $n \cdot m \cdot l$ cost values, calculating $w_M$ based on $w_L$ requires to determine the maximum out of $n \cdot m \cdot l$ sums over $O(m \cdot l)$ cost saving values with complexity $O(n \cdot (m \cdot l)^2)$. The total time complexity of the logical mapping phase is $O(n \cdot (m \cdot l)^2)$.

Now we analyze the complexity of the physical mapping. Due to the regularity of the employed patterns, identifying the qubits associated with a logical variable takes linear time in the number of qubits. Weights on and between qubits can be added in constant time per weight. Calculating the scaling factor $w_B$ for a group $B$ of qubits requires to examine the connections of each qubit in $B$. As each qubit is connected to a constant number of other qubits, the time for calculating $w_B$ is linear in $|B|$. In summary, we must calculate scaling factors for $O(n \cdot (m \cdot l)^2)$ qubits and assign $O(n \cdot (m \cdot l)^2)$ weights. The combined complexity of logical and physical mapping is $O(n \cdot (m \cdot l)^2)$. □

We find that the time complexity of the transformation from MQO problems into qubit weight assignments is a low-order polynomial in the MQO problem dimensions. We have not taken into account the complexity of clustering queries, generating alternative plans for each query, and identifying work overlap. This pre-processing step is however required by other MQO optimization methods as well [95] and its implementation is orthogonal to the selection of optimal plan combinations. If it is initially unclear how many clusters are required then the mapping algorithm can be invoked iteratively for a decreasing number of clusters until the mapping is successful (which is assured for one cluster). Then the time complexity is multiplied by the number of iterations.

## 8.7 Experimental Evaluation

We were granted a limited amount of computation time on a D-Wave 2X adiabatic quantum annealer with over 1000 qubits that is currently located at NASA Ames Research Center in California. We evaluated its performance on MQO problem instances that have been transformed into mathematical formulas as described before.

Our current approach transforms one MQO problem instance into one QUBO problem instance while we might consider approaches mapping one MQO problem into series of QUBO problems in future work. The size of the problems that can be treated by our current approach is inherently limited by the number of available qubits. The formulas established in the last section can be used to calculate the limits on the MQO problem dimensions until which our approach is applicable. It is clear, without performing any experiments, that there are classes of MQO problems that can be treated by existing MQO algorithms (e.g., 500 queries with three plans or more per query [19]) but not on a quantum annealer with 1097 qubits. This is why we focus our experiments on the opposite question: are there also classes of MQO problems where finding the optimal solution requires non-negligible optimization time on commodity computers and where the quantum annealer outperforms existing approaches?

This question is interesting since a positive answer would constitute evidence that future models of the quantum annealer with more qubits can become an interesting alternative to classical MQO optimizers and the number of qubits has so far been steadily doubling from one model to the next. The question is also non-trivial and experiments are required to answer it: while absolute optimization times are expected to be lower for the quantum annealer than for commodity computers when optimizing the problem class that is natively supported by the quantum annealer, the blowup in problem representation size during logical and physical mapping might in principle offset that advantage.

We answer the aforementioned question in the following. Section 8.7.1 describes our experimental setup while Section 8.7.2 describes and discusses our experimental results.

### 8.7.1 Experimental Setup

We use a D-Wave 2X quantum annealer as described in Section 8.2. We use the default time of 129 microseconds per annealing run and 247 microseconds per read-out such that an annealing run with following readout takes 376 microseconds. For each test case, we perform 1000 annealing runs that are partitioned into 10 batches of 100 annealing runs per gauge transformation. A gauge transformation [29] selects for each qubit the physical state representing a one randomly between the two available states. Using multiple gauge transformations reduces the effect of small biases favoring one qubit state over another.

We compare our approach based on quantum annealing against other optimization algorithms that have been recently proposed for MQO: integer linear programming [54], genetic algorithms [19], and iterated hill climbing [53]. We compare against a commercial integer linear programming solver that we use in two ways: we use it to solve MQO problems directly and we use it to minimize the energy formula that the quantum annealer minimizes, too. We use a linear reformulation of the quadratic energy formula that is more suitable for integer programming solvers [47]. We do not compare against classical algorithms that are specialized to the D-Wave hardware, a corresponding benchmark has been released recently [86]. We focus on a comparison between the quantum annealer and classical algorithms that solve MQO problem instances without requiring a transformation that increases the number of problem variables (this puts the quantum annealer at a disadvantage).

Our heuristic algorithms are implemented in Java (while the integer linear programming solver is implemented in C). We use the Java Genetic Algorithms Package[6] in version 3.6.3 with the default configuration which is a genetic algorithm with single point crossover and a top-n selection strategy. The crossover rate is 0.35 and the mutation rate 1/12. We try different population sizes in our experiments. Our hill climbing algorithm iteratively generates plan selections randomly and improves them via hill climbing until a local optimum is reached. We follow good practices for benchmarking Java programs[7] and execute a code warmup of at least 10 seconds for each algorithm before starting the actual benchmark. All Java-based algorithms were implemented in Java 1.7 and executed using the Java HotSpot(TM) 64-Bit Server Virtual Machine version on an iMac with i5-3470S 2.90GHz CPU and 16 GB of DDR3 RAM.

We focus on the core optimization problem and neither perform common sub-expression identification nor query clustering. We consider test cases that map well to the quantum annealer for the reasons outlined before. We vary the number of alternative plans per query and generate test cases for the maximal number of queries that can be represented with the available number of qubits. Each query forms one cluster. The weights between qubits representing different plans for the same query are determined by our mapping scheme. The weights between qubits representing plans of different queries represent cost savings and are chosen randomly.

### 8.7.2 Experimental Results

We compare optimization approaches in terms of how solution quality, measured by the scaled execution cost of the current plan selection, evolves as a function of optimization time. We measure execution cost at regular time intervals, after 1, 10, 100, 1000, $10^4$, and $10^5$

[6]http://jgap.sourceforge.net/
[7]http://www.ibm.com/developerworks/library/j-benchmark1/

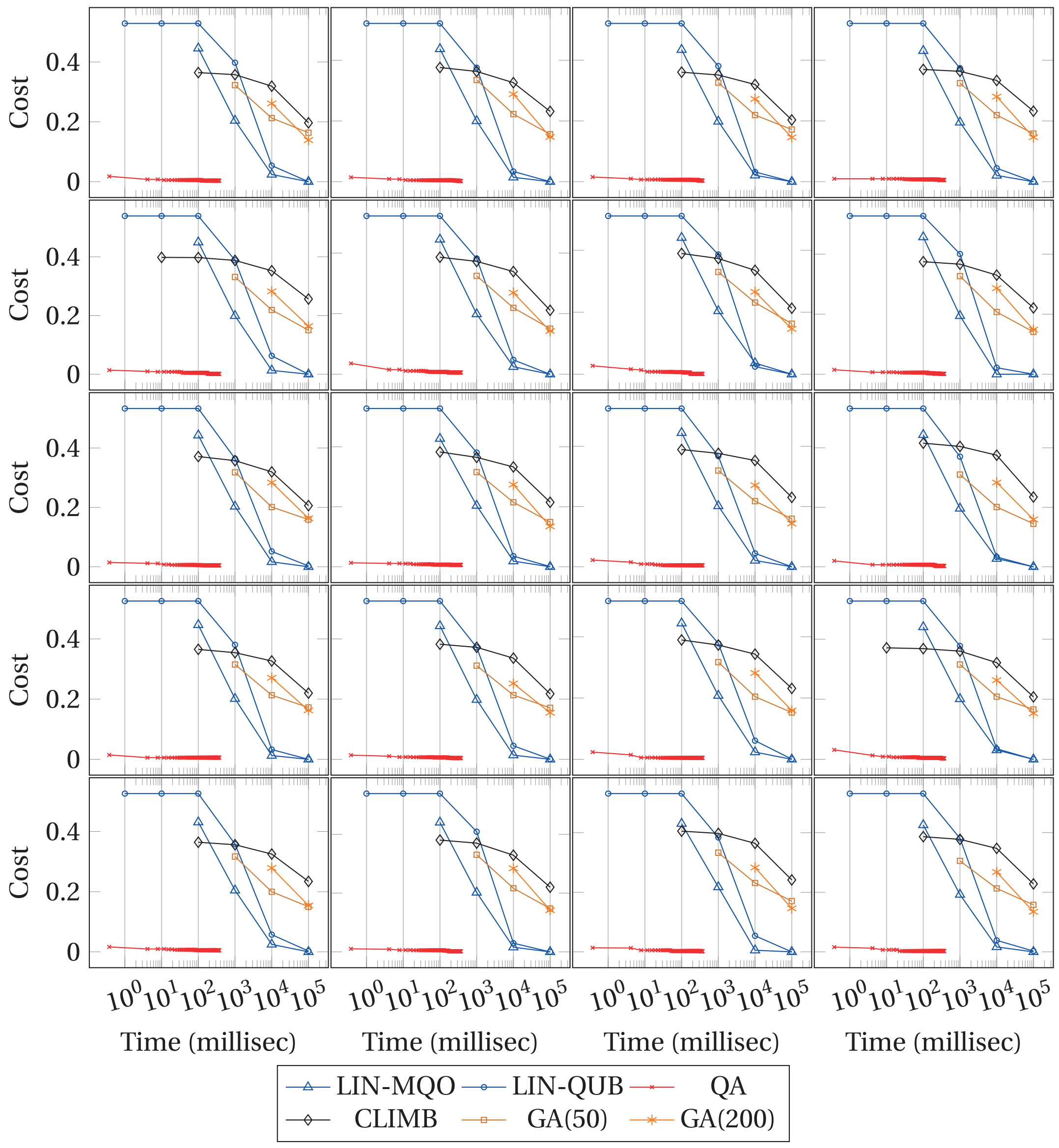


Figure 8.4 – Solution cost as a function of optimization time for 20 MQO problem instances with 537 queries and 2 plans per query.

milliseconds. For the quantum annealer, we report the execution cost of the best solution found after each batch of 10 annealing runs in the following figures (this information is generated by default and using it does not introduce measurement overheads). We consider pure optimization time in the following and do not include pre-processing times required to transform MQO problem instances into linear programs (for the integer programming solver) or quadratic programs (for the quantum annealer). The corresponding pre-processing times, using an unoptimized implementation, were between 112 and 135 milliseconds per test case for the quantum annealer.

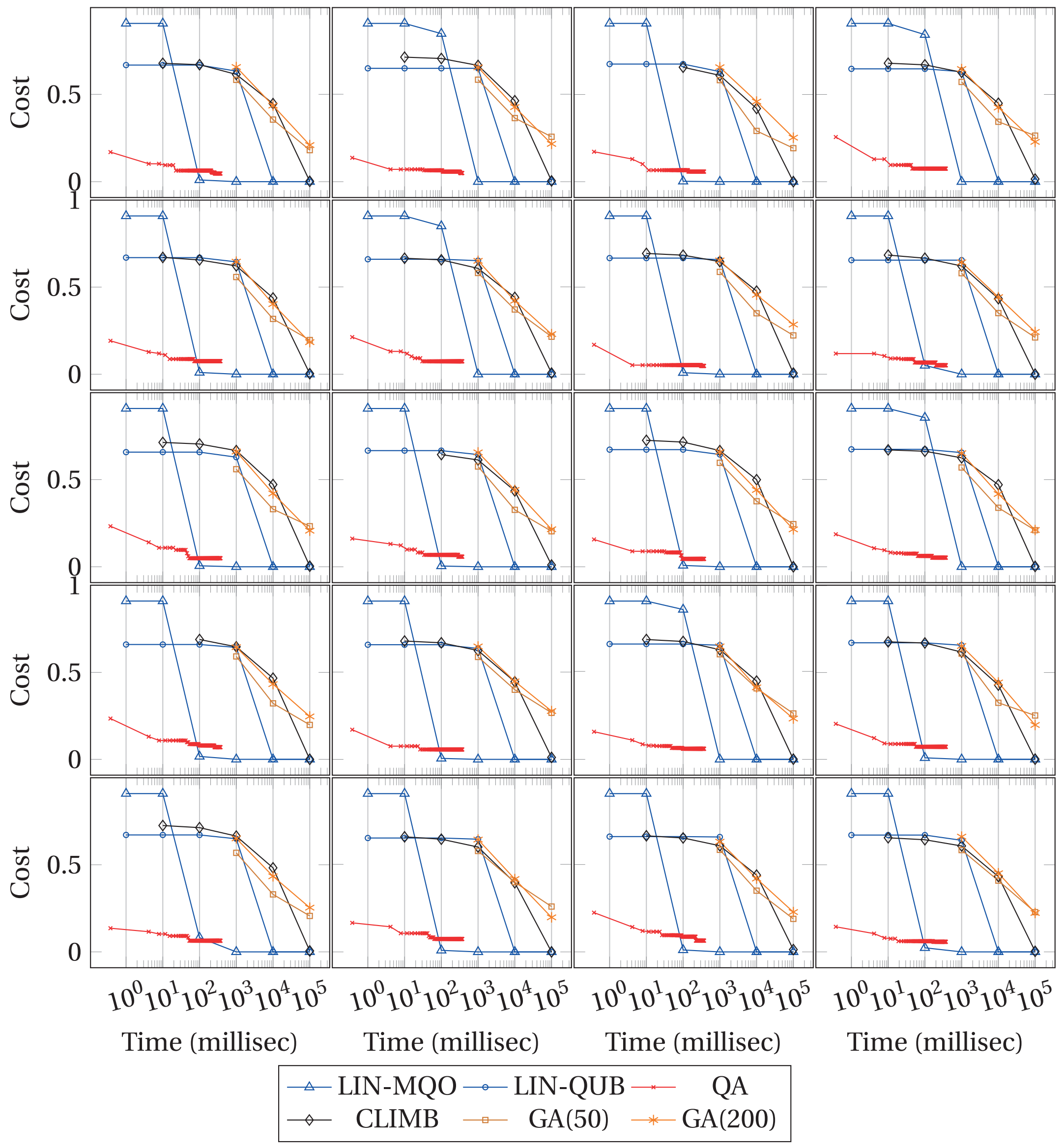


Figure 8.5 – Solution cost as a function of optimization time for 20 MQO problem instances with 253 queries and 3 plans per query.

Figures 8.4 to 8.7 show the performance results for between two and five alternative plans per query and the associated maximal number of queries that can be treated using the available qubits (between 537 queries for two plans and 108 queries for five plans). Note that the x-axis, on which optimization time is represented, is logarithmic. Each figure shows detailed results for each out of 20 test cases. We chose to represent performance results for single test cases to give an intuition about how consistent the performance differences between the compared approaches are. The figure legends use the abbreviations QA for quantum annealer, LIN-MQO for linear solver applied to MQO problem instances, LIN-QUB for linear solver applied to QUBO instances, CLIMB for iterated hill climbing, and GEN(50) and GEN(200) for the genetic

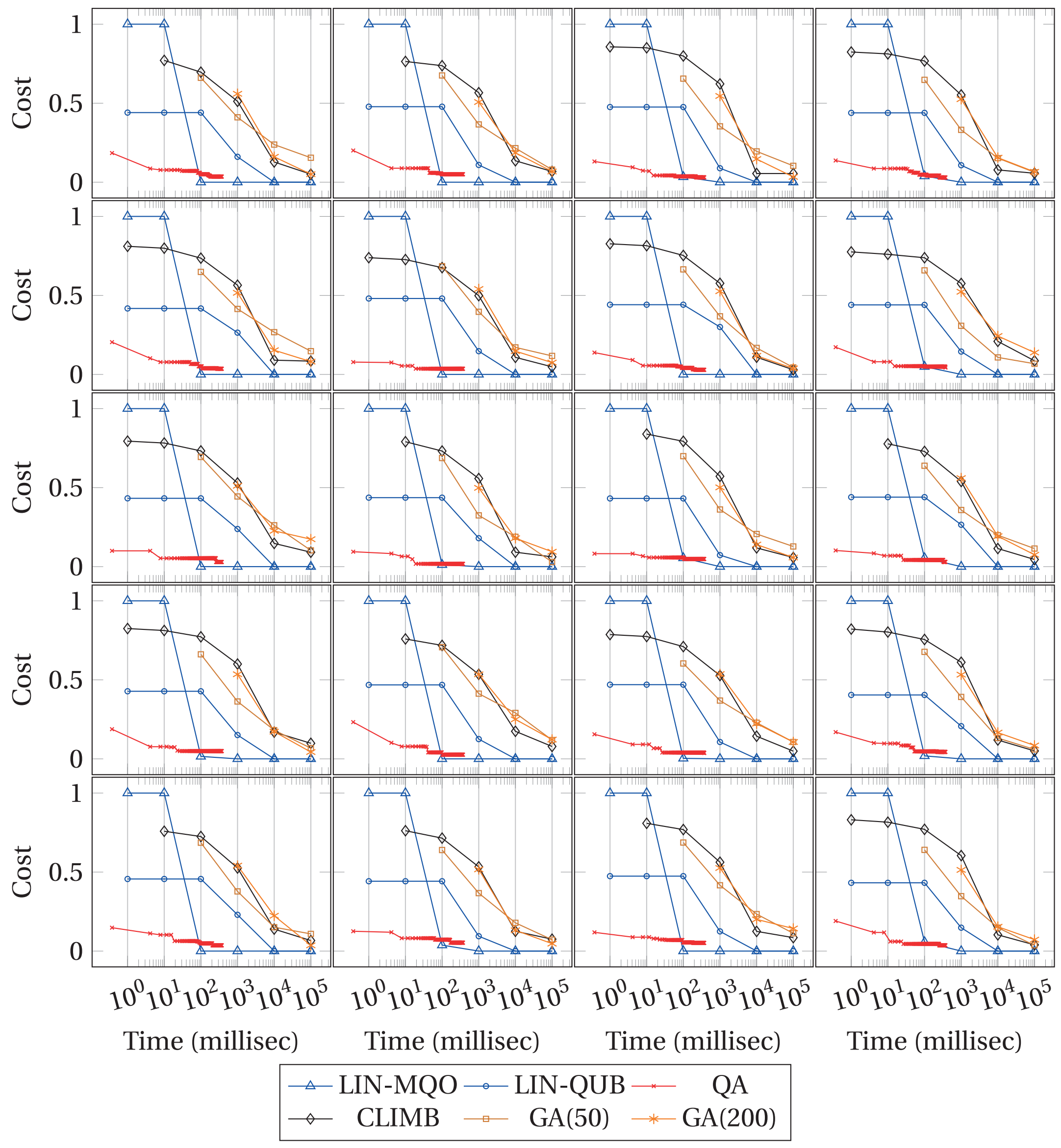


Figure 8.6 – Solution cost as a function of optimization time for 20 MQO problem instances with 140 queries and 4 plans per query.

algorithm with population size 50 and 200 respectively.

We first discuss the results shown in Figure 8.4. The corresponding class of test cases with 537 queries is the hardest class among the ones we consider if judging hardness by the time it takes to find the optimal solution using the linear solver directly on the MQO problem instance (Table 8.1 shows aggregates of the time it takes to find the optimal solution depending on the number of queries).

Among the approaches executed on a classical computer, the integer linear programming solver achieves the best results in the optimization time range between 1 and 100 seconds. The

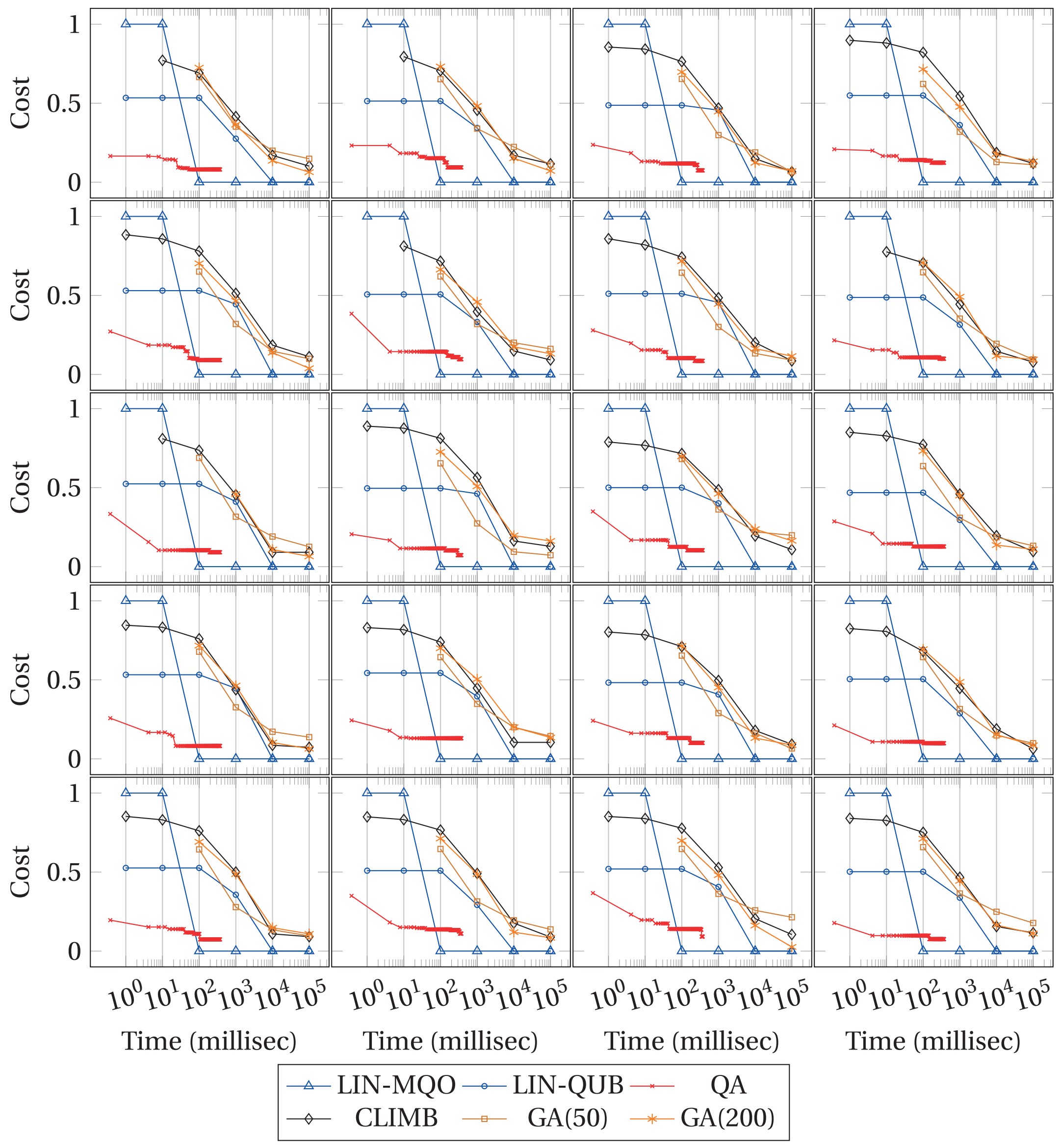


Figure 8.7 – Solution cost as a function of optimization time for 20 MQO problem instances with 108 queries and 5 plans per query.

Table 8.1 – Milliseconds until finding the optimal solution via integer linear programming.

| # Queries | Minimum | Median | Maximum |
|---|---|---|---|
| 537 | 9261 | 25205.5 | 34570 |
| 253 | 129 | 178.5 | 206 |
| 140 | 45 | 128 | 241 |
| 108 | 47 | 48 | 51 |

performance is clearly better when solving the MQO problem directly instead of the QUBO representation that was derived from it. This is to be expected as the MQO representation leads to a smaller search space than the QUBO representation: only the QUBO representation allows to represent invalid solutions where multiple or no plans are selected for some queries which leads to an exponential blowup in search space size in the number of alternative plans per query.

The solutions produced by the randomized algorithms are clearly inferior to the ones found by the linear solver after one second of optimization time. Before that time, the hill climbing algorithm often produces slightly better solutions than the linear solver. Over the long term, the hill climbing algorithm is however beaten by the genetic algorithm. This is intuitive as the hill climbing algorithm is the simplest one among all compared approaches.

All classical approaches have in common that solution quality improves significantly over a time span of several seconds. This is different for the quantum annealer. The execution cost of the solution found after the first annealing run is relatively close to the best execution cost found after 1000 runs with an average cost reduction of 1.5% from run one to run 1000. As the cost of the final solution after 1000 annealing runs is very close to the optimal solution found by the linear solver (with an average cost overhead of 0.4%), this means that the quantum annealer produces good solutions very quickly compared to the other approaches.

More precisely, in 13 out of 20 test cases, the quantum annealer finds a solution after one annealing run (which takes less than half a millisecond) that is better or equivalent to the solutions found by all other approaches after 10 seconds. The best solution obtained during the first 10 annealing runs is in 18 out of 20 test cases at least equivalent to the solutions generated after 10 seconds by the other approaches. The solution returned by the first annealing run is for all test cases at least equivalent to the solutions generated by the other approaches after one second. This shows that there is a range of MQO problems in which the quantum annealer consistently outperforms the other approaches with a speedup of more than factor 1000.

The performance advantage of the quantum annealer over the other approaches gradually decreases as we increase the number of plans per query which decreases the number of queries that can be represented with the available number of qubits. Figures 8.5 to 8.7 show that development. In Figure 8.7, the quantum annealer is superior in the optimization time range up to 10 milliseconds while the linear solver finds the optimal solution in less than 100 milliseconds.

We attribute this effect to two related reasons: First, as shown by our analysis in Section 8.6, increasing the number of alternative plans per query increases the number of qubits required for representing one single logical variable quickly. This means that the search space size of the problems that can be mapped to the quantum annealer decreases. Experimenting with easier problems generally tends to decrease the performance gap between optimization algorithms. On the other side, the ratio between QUBO solutions representing invalid MQO solutions and QUBO solutions representing valid MQO solutions increases exponentially in

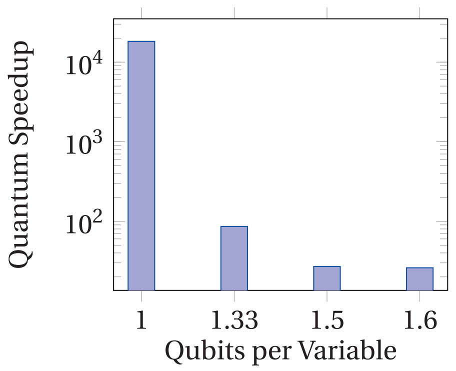


Figure 8.8 – Average speedup for different classes of test cases: having to use more qubits per problem variable decreases the speedup.

the number of plans per result. Hence the drawback of having to work with a reformulation of the original problem becomes more significant as the number of plans increases.

Figure 8.8 shows similar tendencies: it reports the ratio of the time required by the best classical approach to match the result quality produced by the quantum annealer after one sample divided by the time required by the quantum annealer to produce that sample. The figure reports average speedup for each of the four classes of test cases that we generated. We correlate the speedup with the number of qubits required to represent a single problem variable. Again, the speedup decreases quickly once more qubits are required to represent a single problem variable.

In summary, we have identified a class of MQO problems where the quantum annealer in combination with our mapping method outperforms approaches on classical computers in finding near-optimal solutions by several orders or magnitude. This performance advantage decreases however quickly once the problem instances are less convenient to represent as QUBO problems.

## 8.8 Related Work

Our work relates to prior work on MQO, to publications showing how to solve specific problems using a quantum computer, and to experimental evaluations of quantum annealers for specific problem classes.

The MQO problem [119] is a classical database-related optimization problem. The goal of MQO is to reduce execution cost by sharing work among queries. This requires preparatory steps such as identifying common expressions among queries and generating alternative plans for each query [52, 79]. The optimization problem of selecting an optimal combination of plans for execution is orthogonal to the problem of identifying common sub-expressions and generating plans that allow to exploit them. We focus on plan selection.

Various approaches have been proposed for selecting an optimal combination of plans in MQO. The first generation of MQO approaches were branch-and-bound algorithms or based on the A-* algorithm [44, 119, 121]. Such approaches scale only to a limited number of queries [19] which motivates the use of randomized algorithms such as genetic algorithms [19, 53] or efficient greedy heuristics such as hill climbing [46, 53, 83, 99, 114]. Approaches based on integer linear programming [21, 54] have been shown to outperform prior algorithms [54] if the goal is to find optimal MQO solutions. We selected a representative subset of recently proposed MQO approaches for our experimental evaluation. We did not consider approaches that target specific scenarios (e.g., SPARQL processing [95]) or approaches that are based on a different problem model than the one we consider (e.g., representations based on And-Or-Dags [114]).

Our work connects to other publications showing how to solve specific problems on quantum computers, including for instance database search [63], classification [104], calculation of Ramsey numbers [25, 58], some of the classical NP-hard optimization problems [96], fault detection [111], job shop sheduling [143], or protein folding [110]. Authors affiliated with NASA have recently studied how to solve several optimization problems that are relevant in the context of NASA's future deep space missions on an adiabatic quantum annealer [127]. None of the aforementioned publications treats however the problem of MQO. Furthermore, not all of the aforementioned publications feature an experimental evaluation.

One of the first performance evaluations that compares an adiabatic quantum annealer against classical optimizers was published in 2013 by McGeoch et al. [98]. The evaluated quantum annealer is an earlier version of the one we use in our evaluation; our annealer increases the number of qubits by roughly one order of magnitude compared to the version from 2013 which allows to treat significantly larger search spaces. McGeoch et al. compare the quantum annealer against an integer programming solver in terms of the time it takes to find optimal solutions. While the quantum annealer outperforms the integer programming solver by several orders of magnitude, an alternative representation of the integer programming problem has been shown later to decrease that performance gap significantly [47]. We use the optimized representation in our experiments.

A quantum annealer with 512 qubits, the predecessor of the one we experimented with, was recently compared against classical algorithms by multiple groups [71, 85]. The focus of those evaluations was to compare the asymptotic growth of optimization time until an optimal solution is found between the quantum annealer and traditional optimization algorithms. Results by Hen et al. [71] for a class of Ising problems generated without limiting the weights on and between qubits show slight advantages for the D-Wave annealer only for a very small range of test parameters and no speedup for others while results on weight-limited instances [85] show a robust scalability advantage for the quantum annealer.

The focus of our experimental evaluation differs in several ways. First, we focus on the MQO problem and not on Ising problems which are natively supported by the quantum annealer. This is a challenging scenario for the quantum annealer since the approaches we compare against do not suffer from the blowup in search space size when transforming MQO problems into Ising problems. Second, while the other evaluations essentially compare the quantum annealer against hypothetical massively-parallel classical solvers by scaling down the optimization times of classical solvers, we are interested in the raw optimization times realized by existing systems. Finally, while prior evaluations mostly focus on the time until an optimal solution is found, our evaluation is broader as we consider how solution quality evolves as a function of optimization time.

In addition, our work differs from prior evaluations since we use the newest model of the quantum annealer with over 1000 qubits that was very recently released. To the best of our knowledge, the results in this chapter are the first performance results for the D-Wave 2X besides an initial publication by affiliates of the company D-Wave [86].

## 8.9 Conclusion and Outlook

We have shown how the problem of multiple query optimization can be solved using an adiabatic quantum computer. We analyzed our approach formally and evaluated it experimentally, making this one of the first published experimental evaluations of adiabatic quantum annealers with over 1000 qubits. The quantum annealer finds near-optimal solutions faster than various classical optimization approaches in all evaluated scenarios. The speedup reaches up to three orders of magnitude for a subset of evaluated scenarios. Due to the limited number of qubits, we were only able to compare on a relatively narrow range of problem instances.

Our current mapping approach transforms one MQO problem instance into one QUBO problem instance. We will explore approaches that map one MQO problem instance into a series of QUBO problems in future work which should in principle allow to treat larger problem instances than the ones we have considered in our experiments. Our experimental results are specific to MQO. We plan to address other database-specific optimization problems in future work.

# 9 Conclusion and Outlook

Query optimization is a key problem in the context of structured data processing. We must solve the query optimization problem in order to make the execution of declarative queries efficient [64]. In this thesis, we have introduced new query optimization variants (multi-objective parametric query optimization) and corresponding approaches. Thereby we support scenarios that cannot be addressed by prior query optimization algorithms. For query optimization with multiple execution cost metrics (multi-objective query optimization), we propose the first algorithms that handle diverse cost metrics within reasonable amounts of optimization time. We propose a broad portfolio of algorithms addressing different scenarios in terms of query size, user-optimizer interaction model, and optimization platform. For the classical query optimization variants with one execution cost metric, we show how to significantly extend the size of the problem instances for which we can find optimal or near-optimal solutions.

The techniques by which we extend the scope of query optimization can be classified into three broad categories: moving query optimization before run time, relaxing optimality guarantees, and leveraging new optimization software and hardware platforms. We propose one or several techniques in each of those broad categories. Moving query optimization with multiple cost metrics before run time leads to a new problem variant that we discuss in Chapter 4. Relaxing optimality guarantees in multi-objective query optimization leads to approximation schemes (see Chapter 2), incremental algorithms (see Chapter 3), or randomized algorithms (see Chapter 5). Among the software and hardware platforms that we exploit for query optimization are integer linear programming solvers (see Chapter 7), massively parallel clusters (see Chapter 6), and quantum annealers (see Chapter 8).

Note that this list includes approaches that have never been exploited for optimization problems in the database domain (e.g., adiabatic quantum annealing), for query optimization (e.g., massive parallelization), or for specific query optimization variants (e.g., we describe the first randomized algorithm for multi-objective query optimization in this thesis). The proposed approaches cover different scenarios, some of them can be used in combination. In Section 9.1, we provide guidelines for choosing the most appropriate combination of optimization methods for a given scenario. In Section 9.2, we outline how the methods proposed

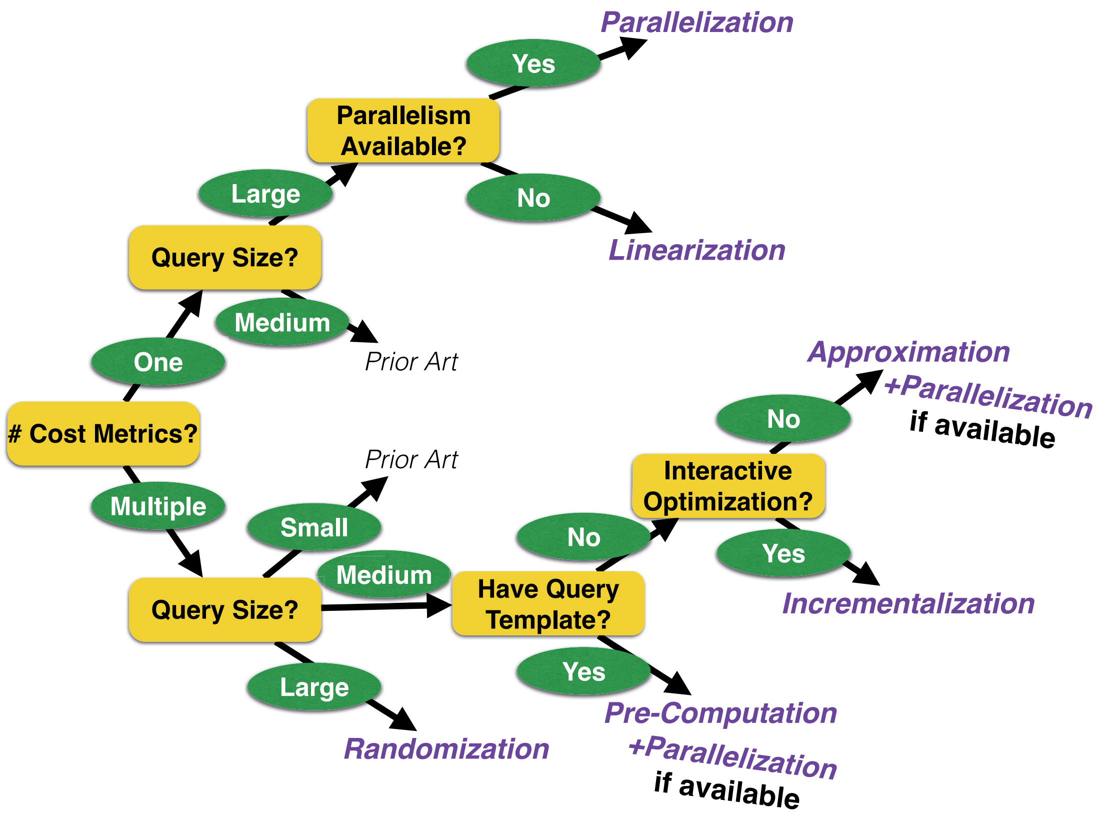


Figure 9.1 – Decision tree for choosing the right query optimization method out of the ones proposed in this thesis.

in this thesis can be integrated into future query optimizers and outline directions for future research.

## 9.1 Choosing the Right Query Optimization Method

The query optimization approaches proposed in this thesis cover various scenarios. In this section, we provide some guidance on how to choose between them. Figure 9.1 shows a decision tree for choosing the right query optimization approach based on the context. In order to obtain a decision tree of reasonable size, we neglect several criteria (e.g., the structure of the join graph) that can influence the performance of the proposed approaches significantly. For that reason, the decision tree in Figure 9.1 should be used with care. It gives a first intuition about which methods are suitable for which scenarios but it should not be followed blindly.

Figure 9.1 should be read from left to right. The most important criterion for selecting a query optimization method is the number of execution cost metrics. Several of the approaches that are proposed in this thesis are specific to the case of multiple execution cost metrics (multi-objective query optimization). We discuss the case of one single execution cost metric first (classical query optimization). In that case, the size of the input query (measured by the number of joined tables) decides on whether or not the methods proposed in this thesis are helpful. Algorithms proposed prior to this thesis (e.g., based on dynamic programming) find optimal query plans for small and medium-sized queries quickly. For large queries, existing optimizers typically switch to randomized or heuristic algorithms that do not guarantee to find an optimal query plan (e.g., the optimizer of the Postgres database system switches to a genetic algorithm starting from 12 joined tables by default[1]).

The approaches proposed in this thesis increase the size of the queries for which optimal plans can be found within a reasonable amount of optimization time. In Chapter 6, we have seen how to exploit large degrees of parallelism to decrease optimization time significantly. If parallelism is available (which nowadays is often the case) then it should be exploited to optimize large queries. Alternatively, software solvers for mixed integer linear programming can be leveraged to solve significantly larger instances than with traditional query optimization algorithms (see Chapter 7). The disadvantage of that approach is that it generates guaranteed near-optimal but not guaranteed optimal query plans since cost functions are linearized (i.e., approximated by linear functions). Using linearization is in each case preferable over using randomized methods that do not offer any worst-case guarantees on the quality of the generated query plans.

Now we discuss the case of multiple execution cost metrics (lower part of Figure 9.1). Again, the query size is an important criterion to select between different optimization methods. Previously proposed algorithms were only able to optimize small queries when considering diverse execution cost metrics. We have shown in Chapter 2 that corresponding algorithms can already consume prohibitive amounts of optimization time for queries joining only six tables (in case of three execution cost metrics). Optimizing medium-sized and large queries with multiple cost metrics requires using the algorithms that are proposed in this thesis.

Randomized algorithms are the only choice for large queries joining hundreds of tables. We have seen a randomized algorithm that is specialized for multi-objective query optimization in Chapter 5. This algorithm does not provide any formal worst-case guarantees on the quality of its output. It performs however well in average and outperforms general purpose algorithms for multi-objective optimization significantly. In the following, we discuss the case of medium-sized queries. Those are queries that are not sufficiently small to apply exhaustive optimization algorithms and not sufficiently large to require randomized optimization.

As shown in Chapter 4, run time optimization can be avoided for medium-sized queries by making query optimization a pre-processing step. This is only possible if queries correspond to a query template that is known before run time. If this requirement is satisfied, we recommend using pre-processing as it leads to minimal optimization overhead at run time. Note that the parallelization method described in Chapter 6 can be used to speed up pre-processing. Similar to the other methods that we propose for multi-objective query optimization on medium-

[1]http://www.postgresql.org/docs/9.2/static/runtime-config-query.html

sized queries, pre-processing does not lead to optimal query plans but to near-optimal query plans. This is due to the fact that cost functions are approximated by piecewise-linear cost functions.

If no query templates are available, the choice of an optimization method depends on the desired interaction between user and optimizer. Interactive optimization is interesting for optimizing long-running queries (or queries that are frequently executed) as it requires the user to invest time during optimization. Chapter 3 describes an incremental algorithm that supports interfaces for interactive query optimization. In principle, the parallelization method from Chapter 6 could be used in combination with incrementalization. The purpose of incrementalization is however to achieve high update rates in interactive interfaces. In that specific context, parallelization might add latency due to communication overhead. For that reason, parallelization seems less beneficial than in other scenarios.

If users specify their preferences before optimization starts instead of interacting with the optimizer then incremental optimization is unnecessary (and should be avoided as it causes overheads). Then the approximation algorithms described in Chapter 2 become the first choice. The parallelization method described in Chapter 6 can be used in combination.

Figure 9.1 focuses on optimization algorithms that optimize one single query. Therefore, it does not include the approach for exploiting adiabatic quantum annealing described in Chapter 8. The latter approach is targeted at scenarios in which a set of queries has to be answered efficiently by merging computation between different queries. This approach requires that a small set of query plans has been identified for each query in the query set. The approaches in Figure 9.1 can be used on each query separately to calculate a small set of query plans with similar cost tradeoffs. In a second step, the approach from Chapter 8 can be used to select the optimal combination of plans. Our technique can only be applied if a quantum annealer is available but corresponding machines are rare. Therefore, in contrast to the other approaches, we see our work in this direction more as a proof of concept.

## 9.2 Outlook and Future Work

Query optimization is a key problem in the area of databases. Nearly all relational database systems employ nowadays cost-based optimizers and the work presented in this dissertation is therefore relevant to all of them. Beyond traditional database systems, the presented techniques are relevant to tools such as Hive [4] and Spark SQ [7] offering SQL-like interfaces on top of frameworks such as Hadoop [3]. Services such as Google's BigQuery [2] and Amazon's RedShift [1] offering SQL processing in the Cloud benefit from the proposed techniques as well.

The applicability of our optimization methods extends beyond tools with SQL interfaces. Popular libraries such as MADlib [5] translate machine learning workflows containing linear algebra expressions into SQL queries. Tools such as Tableau [8] provide visual interfaces but use relational queries in the background. Even information extraction systems such as DeepDive [105] use relational queries during certain extraction stages. All those and other systems and tools can indirectly benefit from approaches for optimizing SQL queries.

Integrating our optimization methods into future query optimizers allows solving problem variants for which no prior algorithms exist or for which prior algorithms need prohibitive amounts of optimization time. Some of our approaches also enable new user-optimizer interaction methods (interactive query optimization) or reduce implementation overhead (by replacing the optimizer core by standard solvers). Current query optimizers often implement a combination of several query optimization algorithms (e.g., the Postgres optimizer uses an exhaustive and a randomized algorithm and chooses between them based on the query size[2]). Future query optimizers might use a combination of the methods described in this thesis. Which methods are most interesting depends on the scenario. In the last section, we have discussed guidelines for selecting optimization methods based on the scenario.

For several reasons, the value of the approaches presented in this thesis is likely to increase further in the future. Growing data sizes motivate flexible provisioning methods such as Cloud computing and crowdsourcing and approximate processing techniques. Cost and quality metrics such as monetary fees, result precision, or recall become important in addition to execution time. Hardware platforms are becoming more heterogeneous which offers more possibilities to trade between execution time and energy consumption. Multitenancy models become more attractive with growing data sizes as moving data away from a central storage location is prohibitively expensive for large data sets. In the context of multitenancy models, tradeoffs between the amount of system resources dedicated to one user and the performance perceived by that user need to be considered. In general, growing data sizes and processing requirements drive advances in processing platforms and processing techniques. But sophisticated processing platforms and techniques often motivate new execution cost metrics and offer new possibilities to trade between them. Thereby, the need for multi-objective query optimization increases.

Data processing systems are nowadays massively parallel and the degree of parallelism keeps growing. This increases also the benefit of parallel query optimization. The approach presented in this thesis is aimed at exploiting massive degrees of parallelism for query optimization. It guarantees nearly skew-free parallelization, the complexity of all serial steps and the amount of network traffic grows only polynomially in the query size. For that reason, this approach seems to be able to exploit the excessive degrees of parallelism that are to be expected in future data processing systems. Systems nowadays increase rather their degree of parallelism than the speed of single processors. In the long term, query optimization has to become massively parallel in order not to become the bottleneck of query evaluation. This makes it likely that future query optimizers will use parallelization approaches that are similar to the one presented in this thesis.

[2]http://www.postgresql.org/docs/9.0/static/planner-optimizer.html

Mixed integer linear programming (MILP) solvers have steadily improved their performance (hardware-independently) over the last decades. Mapping a problem into the MILP formalism is an investment into the future development of MILP solvers, since solver improvements will automatically increase the value of that transformation. We have seen in this thesis how query optimization instances can be expressed as MILP problems. Already today, this transformation allows us to solve larger problem instances than classical query optimization methods. The value of this transformation will however increase further as MILP solvers become more efficient.

Quantum computing is another technology whose potential might increase in the future. Quantum annealers have so far steadily doubled the number of qubits from one machine model to the next. It is currently expected that this growth will continue in the near and medium term. The limited number of available qubits on current machines restricts the applicability of the approach presented in this thesis. As the number of qubits grows, those restrictions will be more and more relaxed. The development of classical computers has started to hit certain physical limitations (e.g., the end of Dennard scaling). This makes it generally interesting to explore new computational paradigms. In this thesis, we made a first step towards leveraging quantum computing for optimization problems that arise in the database domain.

This thesis opens up several directions for future work. Moving multi-objective query optimization before run time leads to a novel problem variant (multi-objective parametric query optimization). We have proposed a first exhaustive algorithm for that variant and analyzed its properties. The high complexity of our algorithm motivates future work on approximate and randomized algorithms for the same problem. For multi-objective query optimization, we have shown that relaxing optimality guarantees is often required to make optimization practical. We have introduced various novel categories of multi-objective query optimization algorithms in terms of their formal guarantees on result quality. In analogy to classical query optimization, where optimization algorithms in different result quality categories have been continuously improved over time, we hope that follow-up work will improve our first algorithms in all those categories as well.

Query optimization algorithms are based on analytical models estimating execution cost of query plans. Cost estimation is problematic, already since execution cost depends on the sizes of intermediate results that occur during the execution of a query plan. Estimating the sizes of those intermediate results is however difficult (e.g., due to correlated query predicates). Research in the area of query optimization usually either improves the state of the art in cost estimation or the state of the art in optimization algorithms (assuming that reliable cost models are available). Most publications in the domain of query optimization fall into one out of those two categories. We reuse previously proposed and evaluated cost models for various execution cost metrics in our work. All contributions made in this thesis fall therefore into the second category (optimization algorithms). The choice of a cost model is however up to a large extent orthogonal to the choice of an optimization algorithm. All our algorithms should therefore be able to benefit from future advances in cost estimation.

The context of query optimization is defined by the query interface, the execution platform, and the optimization platform. Major advances in the state of the art with regards to any of those components change the context of query optimization. Whenever that happens, new research opportunities for query optimization are likely to result.

# Bibliography

[1] Amazon RedShift. http://docs.aws.amazon.com/redshift.

[2] Google BigQuery. https://cloud.google.com/bigquery.

[3] Hadoop. http://hadoop.apache.org/.

[4] Hive. https://hive.apache.org/.

[5] MADlib. http://madlib.net/.

[6] Spark. http://spark.apache.org.

[7] Spark SQL. http://spark.apache.org/docs/latest/sql-programming-guide.html.

[8] Tableau. http://www.tableau.com/.

[9] S. Aaronson. *Quantum Computing Since Democritus*. 2013.

[10] M. Abhirama, S. Bhaumik, and A. Dey. On the stability of plan costs and the costs of plan stability. *VLDB*, 3(1):1137–1148, 2010.

[11] Z. Abul-Basher, Y. Feng, and P. Godfrey. Alternative Query Optimization for Workload Management. In *Database and Expert Systems Applications*, 2012.

[12] 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.

[13] S. Agarwal, B. Mozafari, and A. Panda. BlinkDB: queries with bounded errors and bounded response times on very large data. In *European Conf. on Computer Systems*, pages 29–42, 2013.

[14] T. Albash, W. Vinci, A. Mishra, P. a. Warburton, and D. a. Lidar. Consistency tests of classical and quantum models for a quantum annealer. *Physical Review A*, 91(4), 2015.

[15] 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.

[16] R. Babbush, P. J. Love, and A. Aspuru-Guzik. Adiabatic quantum simulation of quantum chemistry. *Scientific Reports*, 4, 2014.

[17] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, pages 119–130, 2005.

[18] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. In *SIGMOD*, pages 107–118, New York, New York, USA, 2005. ACM Press.

[19] M. A. Bayir, I. H. Toroslu, and A. Cosar. Genetic algorithm for the multiple-query optimization problem. *IEEE Transactions on Systems, Man and Cybernetics*, 37(1):147–153, 2007.

[20] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In *PODS*, pages 212–223, 2014.

[21] L. Bellatreche and S.-a. B. Senouci. SONIC: scalable multi-query optimization. In *Database and Expert Systems Applications*, pages 278–292. 2013.

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

[23] K. Bennett, M. Ferris, and Y. Ioannidis. *A genetic algorithm for database query optimization.* 1991.

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

[25] Z. Bian, F. Chudak, W. Macready, L. Clark, and F. Gaitan. Experimental determination of Ramsey Numbers. *Physical Review Letters*, 111(13), 2013.

[26] J. Bisschop. Integer Linear Programming Tricks. In *AIMMS: Optimization Modeling*, page 75ff. 215.

[27] R. E. Bixby. A Brief History of Linear and Mixed-Integer Programming Computation. *Documenta Mathematica*, pages 107–121, 2012.

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

[29] S. Boixo, T. Albash, F. M. Spedalieri, N. Chancellor, and D. a. Lidar. Experimental signature of programmable quantum annealing. *Nature communications*, 4:2067, 2013.

[30] S. Boixo, T. F. Rønnow, S. V. Isakov, Z. Wang, D. Wecker, D. A. Lidar, J. M. Martinis, and M. Troyer. Evidence for quantum annealing with more than one hundred qubits. *Nature Physics*, 10(3):218–224, 2014.

[31] N. Bruno. Polynomial heuristics for query optimization. In *ICDE*, pages 589–600, 2010.

[32] N. Bruno and R. V. Nehme. Configuration-Parametric Query Optimization for Physical Design Tuning. *SIGMOD*, 2008.

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

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

[35] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.

[36] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *PODS*, pages 255–265, 1995.

[37] Y. Chen and C. Yin. Graceful Degradation for Top-Down Join Enumeration via similar sub-queries measure on Chip Multi-Processor. *Applied Mathematics and Information Sciences*, 941(3):935–941, 2012.

[38] V. Choi. Minor-embedding in adiabatic quantum computation: I. The parameter setting problem. *Quantum Information Processing*, 7(5):193–209, 2008.

[39] V. Choi. Minor-embedding in adiabatic quantum computation: II. Minor-universal graph design. *Quantum Information Processing*, 10(3):343–353, 2011.

[40] F. Chu, J. Halpern, and J. Gehrke. Least Expected Cost Query Optimization: What can we Expect? *SIGMOD*, 2002.

[41] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*, pages 54–67, 1995.

[42] C. a. Coello Coello. Evolutionary multi-objective optimization: a historical view of the field. *Computational Intelligence Magazine*, 1(1):28–36, 2006.

[43] R. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD*, pages 150–160, 1994.

[44] A. Cosar, E. Lim, and J. Srivastava. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In *Information and Knowledge Management*, pages 433–438, 1993.

[45] O. Cure and G. Blin. *RDF Database Systems: Triples Storage and SPARQL Query Processing*. 2014.

[46] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 66(4):728–762, 2003.

[47] S. Dash. A note on QUBO instances defined on Chimera graphs. *arXiv preprint arXiv:1306.1202*, 2013.

[48] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. *Parallel Problem Solving from Nature PPSN VI*, pages 849–858, 2000.

[49] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[50] V. S. Denchev, S. Boixo, S. V. Isakov, N. Ding, R. Babbush, V. Smelyanskiy, J. Martinis, and H. Neven. What is the computational value of finite range tunneling? *arXiv preprint arXiv:1512.02206*, 2015.

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

[52] S. Diego, F.-c. F. Chen, and M. H. Dunham. Common subexpression processing in multiple-query processing. *Knowledge and Data Engineering*, 10(3):493–499, 1998.

[53] T. Dokeroglu, M. A. Bayir, and A. Cosar. Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries. *Applied Soft Computing*, 30:72–82, 2015.

[54] T. Dokeroglu, M. A. Bayır, and A. Cosar. Integer linear programming solution for the multiple query optimization problem. In *Information Sciences and Systems*, pages 51–60. 2014.

[55] T. Erlebach, H. Kellerer, and U. Pferschy. Approximating multiobjective knapsack problems. *Management Science*, 48(12), 2002.

[56] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser. Quantum computation by adiabatic evolution. *arXiv preprint quant-ph/0001106*, 2000.

[57] T. Flach. Optimizing Query Execution to Improve the Energy Efficiency of Database Management Systems. Technical report, 2010.

[58] F. Gaitan and L. Clark. Ramsey numbers and adiabatic quantum computing. *Physical Review Letters*, 108(1):010501, 2012.

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

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

[61] M. Garofalakis and Y. Ioannidis. Multi-Dimensional Resource Scheduling for Parallel Queries. In *SIGMOD*, 1996.

[62] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD*, pages 358–366, 1989.

[63] L. Grover. A fast quantum mechanical algorithm for database search. In *Symposium on the Theory of Computing*, pages 212–219, 1996.

[64] A. Gubichev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *VLDB*, 9(3):204–215, 2015.

[65] S. Guha, D. Gunopoulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, 2003.

[66] P. Haas. Speeding Up DB2 UDB Using Sampling. *The IDUG Solutions Journal*, pages 1–10, 2003.

[67] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing query optimization. In *VLDB*, pages 188–200, 2008.

[68] W.-S. Han and J. Lee. Dependency-aware reordering for parallelizing query optimization in multi-core CPUs. In *SIGMOD*, pages 45–58, 2009.

[69] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, 1997.

[70] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. *SIGMOD*, 22(2):267–276, 1993.

[71] I. Hen, J. Job, J. Job, M. Troyer, and D. Lidar. Probing for quantum speedup in spin glass problems with planted solutions. *arXiv preprint arXiv:1502.01663*, 2015.

[72] A. Hulgeri. *Parametric Query Optimization*. PhD thesis, 2004.

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

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

[75] Y. Ioannidis and Y. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *SIGMOD*, pages 168–177, 1991.

[76] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD Record*, volume 19, pages 312–321, 1990.

[77] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD*, pages 312–321, 1990.

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

[79] M. Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*, pages 191–205. 1985.

[80] V. Kaibel and M. Pfetsch. Some algorithmic problems in polytope theory. *Algebra, Geometry and Software Systems*, 1, 2003.

[81] S. Kambhampati, U. Nambiar, Z. Nie, and S. Vaddi. Havasu: A Multi-Objective, Adaptive Query Processing Framework for Web Data Integration. *ASU CSE*, 2002.

[82] R. Kaushik, C. Ré, and D. Suciu. General database statistics using entropy maximization. In *Database Programming Languages*, pages 84–99. 2009.

[83] A. Kementsietsidis, A. Kementsietsidis, F. Neven, F. Neven, D. V. D. Craen, D. V. D. Craen, S. Vansummeren, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. In *VLDB*, pages 16–27, 2008.

[84] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. *SIGMOD Record*, 23(2):336–347, 1994.

[85] A. D. King. Performance of a quantum annealer on range-limited constraint satisfaction problems. *arXiv preprint arXiv:1502.02098*, 2015.

[86] J. King, S. Yarkoni, M. M. Nevisi, J. P. Hilton, and C. C. Mcgeoch. Benchmarking a quantum annealing processor with the time-to-target metric. *arXiv preprint arXiv:1508.05087*, 2015.

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

[88] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. E. Ioannidis. Schedule Optimization for Data Processing Flows on the Cloud. In *SIGMOD*, 2011.

[89] C. Klymko, B. D. Sullivan, and T. S. Humble. Adiabatic quantum programming: minor embedding with hard faults. *Quantum Information Processing*, 13(3):709–729, 2014.

[90] D. Kossmann, F. Ramsak, S. Rost, and Others. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. In *VLDB*, 2002.

[91] D. Kossmann and K. Stocker. Iterative Dynamic Programming: a New Class of Query Optimization Algorithms. *Trans. on Database Systems*, 1(212), 2000.

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

[93] T. Lanting, a. J. Przybysz, a. Y. Smirnov, F. M. Spedalieri, M. H. Amin, a. J. Berkley, R. Harris, F. Altomare, S. Boixo, P. Bunyk, N. Dickson, C. Enderud, J. P. Hilton, E. Hoskinson, M. W. Johnson, E. Ladizinsky, N. Ladizinsky, R. Neufeld, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, S. Uchaikin, a. B. Wilson, and G. Rose. Entanglement in a quantum annealing processor. *Physical Review X*, 4(2):021041, 2014.

[94] J. A. Lawrence and B. A. Pasternack. *Applied Management Science*. 1997.

[95] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for SPARQL. In *ICDE*, pages 666–677, 2012.

[96] A. Lucas. Ising formulations of many NP problems. *Frontiers in Physics*, 2(5), 2014.

[97] R. Marinescu. Efficient approximation algorithms for multi-objective constraint optimization. *Algorithmic Decision Theory*, 2011.

[98] C. C. Mcgeoch, C. Wang, B. Bc, M. Thom, and D. Walliman. Experimental evaluation of an adiabiatic quantum system for combinatorial optimization. In *International Conference on Computing Frontiers*, 2013.

[99] H. Mistry, P. Roy, K. Ramamritham, and S. Sudarshan. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Record*, 30(2):307–318, 2000.

[100] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006.

[101] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, pages 9–12, 2008.

[102] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[103] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. *VLDB*, pages 91–102, 1992.

[104] H. Neven and V. Denchev. Training a binary classifier with the quantium adiabatic algorithm. *arXiv preprint arXiv:0811.0416*, 2008.

[105] F. Niu, C. Zhang, C. Ré, and J. Shavlik. DeepDive: Web-scale knowledge-base construction using statistical learning and inference. *CEUR Workshop Proceedings*, 884:25–28, 2012.

[106] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *VLDB*, pages 314–325, 1990.

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

[108] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDEW*, pages 442–449, 2007.

[109] A. Perdomo-Ortiz, N. Dickson, M. Drew-Brook, G. Rose, and A. Aspuru-Guzik. Finding low-energy conformations of lattice protein models by quantum annealing. *Scientific reports*, 2(1):571, 2012.

[110] A. Perdomo-Ortiz, N. Dickson, M. Drew-Brook, G. Rose, and A. Aspuru-Guzik. Finding low-energy conformations of lattice protein models by quantum annealing. *Scientific reports*, 2, 2012.

[111] A. Perdomo-Ortiz, J. Fluegemann, S. Narasimhan, R. Biswas, and V. Smelyanskiy. A quantum annealing approach for fault detection and diagnosis of graph-based systems. *The European Physical Journal Special Topics*, 224(1):131–148, 2015.

[112] A. J. Quiroz. Fast random generation of binary, t-ary and other types of trees. *Journal of Classification*, 6(1):223–231, 1989.

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

[114] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

[115] H. Samet. *The Design And Analysis Of Spatial Data Structures*. 1990.

[116] 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.

[117] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *KDE*, 2(2):262–266, 1990.

[118] T. K. Sellis. Multiple Query Optimization, 1988.

[119] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1):23–52, 1988.

[120] H. Shang and M. Kitsuregawa. Skyline operator on anti-correlated distributions. *VLDB*, 6(9):649–660, 2013.

[121] K. Shim, T. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *DKE*, 12(2):197–222, 1994.

[122] S. W. Shin, G. Smith, J. a. Smolin, and U. Vazirani. How “Quantum” is the D-Wave Machine? *arXiv preprint arXiv:1401.7087*, 2014.

[123] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Foundations of Computer Science*, pages 124–134, 1994.

[124] A. Simitsis, P. Vassiliadis, and T. Sellis. State-Space Optimization of ETL Workflows. *Trans. on KDE*, 17(10), 2005.

[125] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing Analytic Data Flows for Multiple Execution Engines. *SIGMOD*, 2012.

[126] A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos. Optimizing ETL workflows for fault-tolerance. In *ICDE*, pages 385–396. Ieee, 2010.

[127] V. Smelyanskiy, E. G. Rieffel, S. I. Knysh, C. P. Williams, M. W. Johnson, M. C. Thom, W. G. Macready, and K. L. Pudenz. A near-term quantum computing approach for hard computational problems in space exploration. *arXiv preprint arXiv:1204.2821*, 2012.

[128] J. Smolin and G. Smith. Classical signature of quantum annealing. *arXiv preprint arXiv:1305.4904*, 2013.

[129] M. a. Soliman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, R. Baldwin, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, and F. Rahman. Orca: A modular query optimizer architectur for big data. In *SIGMOD*, pages 337–348, 2014.

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

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

[132] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

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

[134] A. Swami. Optimization of large join queries: combining heuristics and combinatorial techniques. *SIGMOD*, pages 367–376, 1989.

[135] A. Swami and A. Gupta. Optimization of large join queries. In *SIGMOD*, pages 8–17, 1988.

[136] TPC. TPC-H Benchmark, 2013.

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

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

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

[140] B. Vance and D. Maier. Rapid Bushy Join-Order Optimization with Cartesian Products. *SIGMOD*, 1996.

[141] B. Vance and D. Maier. Rapid bushy join-order optimization with Cartesian products. *SIGMOD*, 25(2):35–46, 1996.

[142] D. Venturelli, S. Mandrà, S. Knysh, B. O'Gorman, R. Biswas, and V. Smelyanskiy. Quantum optimization of fully-connected spin glasses. *arXiv preprint arXiv:1406.7553*, 2014.

[143] D. Venturelli, D. J. J. Marchand, and G. Rojo. Quantum annealing implementation of job-shop scheduling. *arXiv preprint arXiv:1506.08479*, 2015.

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

[145] F. M. Waas and J. M. Hellerstein. Parallelizing extensible query optimizers. In *SIGMOD*, page 871, 2009.

[146] 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.

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

[148] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

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

[150] E. Zitzler and L. Thiele. Performance assessment of multiobjective optimizers: An analysis and review. *Evolutionary Computation*, 7(2):117–132, 2003.

[151] W. Zuo, Y. Chen, F. He, and K. Chen. Optimization Strategy of Top-Down Join Enumeration on Modern Multi-Core CPUs. *Journal of Computers*, 6(10):2004–2012, oct 2011.

# Immanuel Trummer

Chemin de Boston, 9B
1004 Lausanne
Switzerland

immanuel.trummer@gmail.com
www.itrummer.org
*Revised 3/2016*

## INTERESTS

My research focuses on optimization problems that arise in the context of big data analytics. In particular, I have studied various generalizations of the classical Query Optimization problem. Those generalizations are necessary in order to accurately model the capabilities of modern query execution platforms. I am also exploring the potential of Quantum Computing for solving analytics related optimization problems. This research branch is based on a grant giving me access to a D-Wave 2X adiabatic quantum annealer. Beyond optimization, I am interested in Text Mining and machine learning.

## EDUCATION

**Ecole Polytechnique Fédérale de Lausanne (EPFL)** 2010-2016 (expected)
PhD in Computer Science
Advisor: Christoph Koch

**University of Stuttgart & Ecole Centrale de Nantes** 2003-2010
Double Diploma in Computer Science & Engineering
*Obtained with Distinction - Ranked among top five students*

## AWARDS & HONORS

- Selected for ACM SIGMOD Research Highlight Award 2015
- Invitation to publish in "Best of VLDB 2015" (VLDB Journal)
- Google European PhD Fellowship in structured data analysis
- USRA grant for accessing a quantum annealer (machine price: $15 million)
- EPFL IC Teaching Assistant Award 2015
- Scholarship of the German National Academic Foundation
- First graduation prize by the Computer Science Forum Stuttgart
- Scholarship for Academic Excellence by the University of Stuttgart
- Scholarship for the TIME double degree program
- 2nd prize at German national music competition

## RESEARCH EXPERIENCE

**Graduate Student Researcher** 2010-2016
*EPFL, DATA Lab (until 2013: AI Lab)* Lausanne, Switzerland

Traditional query optimization, multi-objective query optimization, multi-objective parametric query optimization, and probably approximately optimal query optimization:

- I unified the research branches of parametric query optimization and of multi-objective query optimization by introducing the MULTI-OBJECTIVE PARAMETRIC QUERY OPTIMIZATION problem. The corresponding paper was invited for publication as ACM SIGMOD Research Highlight.
- I developed a decomposition method for the query optimization problem that allows to solve it by MASSIVE PARALLELIZATION using large clusters with hundreds of nodes.
- I developed APPROXIMATION SCHEMES for multi-objective query optimization allowing to gradually trade optimization time for query plan optimality guarantees.
- I developed an INCREMENTAL ALGORITHM for multi-objective query optimization that enables users to find their preferred cost tradeoff in an interactive process.
- I developed a REDUCTION from query optimization to mixed integer linear programming allowing to leverage integer programming solver implementations for query optimization.
- I introduced PROBABLY APPROXIMATELY OPTIMAL query optimization which models situations in which query optimizers need to estimate predicate selectivity via sampling.

- I developed a RANDOMIZED ALGORITHM for multi-objective query optimization that is tailored to this problem and handles significantly larger queries than prior approaches.

Machine learning and text mining:

- In collaboration with researchers from Google Mountain View, I have developed a system that mines SUBJECTIVE ENTITY-PROPERTY ASSOCIATIONS from Web text at a very large scale. This system learns entity type and property specific user behavior models in an unsupervised manner and exploits them to interpret collected text fragments more reliably.

Quantum computing:

- I am experimenting with a D-Wave 2X adiabatic quantum annealer, located at NASA Ames Research Center in California. I evaluate the long-term potential of QUANTUM COMPUTING for solving analytics related optimization problems.
- I have shown that the MULTIPLE QUERY OPTIMIZATION problem can be solved on the D-Wave quantum annealer with speedups of up to three orders of magnitude compared to traditional approaches.

**Undergraduate Researcher** 2003-2010
University of Stuttgart Stuttgart, Germany

Cloud computing:

- I developed a method for finding OPTIMAL PROVISIONING strategies for Cloud applications using constraint programming.

## PUBLICATIONS

**Journal Articles**

- Immanuel Trummer, Christoph Koch.
  Multi-objective parametric query optimization.
  Published as *ACM SIGMOD Research Highlight 2015.*
- Immanuel Trummer, Christoph Koch.
  Multi-objective parametric query optimization.
  "Best of VLDB 2015" *(VLDB Journal)* – Accepted subject to minor revisions.
- Immanuel Trummer, Boi Faltings, Walter Binder.
  Multi-objective quality-driven service selection –
  A fully polynomial time approximation scheme.
  *TSE*, 2014,40(2):167-191.

**Conference Publications**

- Immanuel Trummer, Christoph Koch.
  A fast randomized algorithm for multi-objective query optimization.
  *SIGMOD 2016.*
- Immanuel Trummer, Christoph Koch.
  Multiple query optimization on the D-Wave 2X adiabatic quantum computer.
  *VLDB 2016.*
- Immanuel Trummer, Christoph Koch.
  Parallelizing query optimization on shared-nothing architectures.
  *VLDB 2016.*
- Immanuel Trummer, Christoph Koch.
  An incremental anytime algorithm for multi-objective query optimization.
  Talk Recording: `https://www.youtube.com/watch?v=J54gVIt9UAo`
  *SIGMOD 2015.*
- Immanuel Trummer, Alon Halevy, Hongrae Lee, Sunita Sarawagi, Rahul Gupta.
  Mining subjective properties on the Web.
  Talk Recording: `https://www.youtube.com/watch?v=a9RYBydQRXA`
  *SIGMOD 2015.*

- Immanuel Trummer, Christoph Koch.
  Multi-objective parametric query optimization.
  Talk Recording: `https://www.youtube.com/watch?v=hO3IaSfFtJY`
  *VLDB 2015.*
  Invited to "Best of VLDB 2015" VLDBJ Issue.
  Selected as ACM SIGMOD Research Highlight.
- Immanuel Trummer, Christoph Koch.
  Approximation schemes for many-objective query optimization.
  *SIGMOD 2014.*
- Mehdi Riahi, Thanasis Papaioannou, Karl Aberer, Immanuel Trummer.
  Utility-driven data acquisition in participatory sensing.
  *EDBT 2013.*
- Immanuel Trummer, Boi Faltings.
  Optimizing the tradeoff between discovery, composition,
  and execution cost in service composition.
  *ICWS 2011.*
- Immanuel Trummer, Boi Faltings.
  Dynamically selecting composition algorithms
  for economical composition as a service.
  *ICSOC 2011.*
- Immanuel Trummer, Frank Leymann, Ralph Mietzner, Walter Binder.
  Cost-optimal outsourcing of applications into the clouds.
  *CloudCom 2010.*

**Patents**

- Immanuel Trummer, Boi Faltings.
  A method for multi-objective quality-driven service selection.
  *US Patent Application 13/670,864 (US) 2012.*

**Theses**

- Immanuel Trummer.
  Cost-optimal provisioning of cloud applications.
  *Diploma Thesis*, 2010.

**Technical Reports**

- Immanuel Trummer, Christoph Koch.
  Solving the join ordering problem via mixed integer linear programming.
  `http://arxiv.org/pdf/1511.02071v1.pdf`, 2015.
- Immanuel Trummer, Christoph Koch.
  Probably approximately optimal query optimization.
  `http://arxiv.org/pdf/1511.01782v1.pdf`, 2015.

## TEACHING & MENTORING

**Teaching Assistant**, Big Data — Fall 2014

I mentored four teams of students (eight to ten students per group) throughout their semester course projects. The projects I supervised ranged from the design and development of a system for crowdsourced SQL processing to the computer-based analysis of language shift in a corpus containing newspaper articles spanning nearly 200 years. One of my teams won the course-internal competition for the best project.

I received the EPFL Teaching Assistant Award 2015 in appreciation of my work as a teaching assistant for the Big Data course.

**Teaching Assistant**, Introduction to Computer Science — Spring 2014

My task was to design exercises and exams and to supervise exercise sessions in the context of a course introducing bachelor students to various fields in computer science.

**Teaching Assistant**, Linear Algebra — Fall 2013
My task was to supervise exercise sessions in the context of a course introducing basic linear algebra to bachelor students.

**Teaching Assistant**, Advanced Databases — Spring 2013
My task was to design homeworks and exercises and to supervise exercise sessions in the context of a master-level course introducing advanced topics in databases.

**Teaching Assistant**, C++ Programming — Fall 2012
My task was to design exams and to supervise exercise sessions in the context of a course introducing bachelor students to C++ programming.

**Teaching Assistant**, Artificial Intelligence — Spring 2012
My task was to design exams and to supervise exercise sessions in the context of a course introducing bachelor students to artificial intelligence.

**Teacher at Konzept AG**, Computer Science — Fall 2009-Spring 2010
My task was to coach a small group of IT apprentices who had encountered difficulties during their apprenticeship, making them eligible for state-sponsored auxiliary lessons. I had complete freedom in the design of the course. I selected the topics of the course in agreement with the students and designed the course material as well as the exercises myself.

## INDUSTRIAL EMPLOYMENT

**Google** — 5/2014-9/2014
Intern, Web Answers & Web Tables — Mountain View, USA
I designed and implemented a system for mining subjective property associations from Web text. The resulting system was successfully used to infer billions of entity-property associations from a Web snapshot and led to a publication at SIGMOD 2015. I was supervised by Alon Halevy and Hongrae Lee.

**IBM** — 7/2007-10/2007
Intern, Extreme Blue Program — Böblingen, Germany
Within a team of four students, I took part in the design and implementation of a system for RFID-based temperature tracking during DHL transports.

**Agricultural Ministry of Mali** — 4/2006-6/2007
Software Developer — Nantes, France
I designed and implemented an IT system (database, Web interface and standalone client) for the storage and treatment of agricultural data. This project was commissioned by the agricultural ministry of Mali to the Junior Enterprise at Ecole Centrale de Nantes.

**Alcatel** — 7/2006-8/2006
Intern, R&D Division — Paris, France
I designed and implemented a Web interface for network surveillance.

## LANGUAGES

| | |
|---|---|
| **German** | Native speaker. |
| **English** | Published and taught in English for six years. |
| **French** | Eight years of studies in francophone countries; taught courses in French. |

**References are available upon request.**