Specification and Optimization of Analytical Data Flows

vorgelegt von
M.Comp.Sc
Fabian Hüske
aus Soest

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr. Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Uwe Nestmann
Gutachter:	Prof. Dr. Volker Markl
	Prof. Dr. Odej Kao
	Prof. Dr. Rainer Gemulla

Datum der wissenschaftlichen Aussprache: 14. Dezember 2015

Berlin 2016
D 83

Acknowledgements

First and foremost, I would like to thank my advisor Volker Markl. I met Volker the first time when I was an undergraduate student and pursuing a summer internship at the IBM Almaden Research Center about ten years ago. This internship sparked my interest for computer science research and database system building in particular. Since then, Volker has been my most important mentor and encouraged and helped me to follow the academic path. When Volker became a professor at TU Berlin he gave me the prime opportunity to join him as a Ph.D. student. In the course of my Ph.D. studies Volker gave me both, guidance and freedom, to conduct my research and lots of very valuable advice. Thank you Volker.

During my time as a Ph.D. student, many people accompanied and worked with me. I would like to express my sincere gratitude to Stephan Ewen, Kostas Tzoumas, and Daniel Warneke. Working with Stephan, Kostas, and Daniel on the Stratosphere project and building a marvelous parallel data processing system has taught me so much. I would like to thank my colleagues Alexander Alexandrov, Christoph Boden, Max Heimel, Aljoscha Krettek, Marcus Leich, Alexander Löser, Matthias Ringwald, and Sebastian Schelter from the Database Systems and Information Management Group of TU Berlin, who helped me through advice, discussions, and collaboration. Furthermore, I would like to thank my co-authors and fellow Ph.D. students Arvid Heise, Mathias Peters, Astrid Rheinländer, and Matthias Sax for many ideas, discussions, and working with me.

I would also like to express my gratitude to Odej Kao and Rainer Gemulla for serving as my committee members.

Finally, I would like to thank my family. My parents Ilona and Friedel have always supported me in any possible way. When I moved to Berlin to start my Ph.D. in June 2008, I was looking for a room and met Henrike who subleased a room to me. Seven years have passed since then and we are still living in the same apartment. In the meantime we have become parents of three wonderful children. Thank you for everything you have done for me.

Abstract

In the past, the majority of data analysis use cases was addressed by aggregating relational data. Since a few years, a trend is evolving, which is called "Big Data" and which has several implications on the field of data analysis. Compared to previous applications, much larger data sets are analyzed using more elaborate and diverse analysis methods such as information extraction techniques, data mining algorithms, and machine learning methods. At the same time, analysis applications include data sets with less or even no structure at all. This evolution has implications on the requirements on data processing systems. Due to the growing size of data sets and the increasing computational complexity of advanced analysis methods, data must be processed in a massively parallel fashion. The large number and diversity of data analysis techniques as well as the lack of data structure determine the use of user-defined functions and data types. Many traditional database systems are not flexible enough to satisfy these requirements. Hence, there is a need for programming abstractions to define and efficiently execute complex parallel data analysis programs that support custom user-defined operations.

The success of the SQL query language has shown the advantages of declarative query specification, such as potential for optimization and ease of use. Today, most relational database management systems feature a query optimizer that compiles declarative queries into physical execution plans. Cost-based optimizers choose from billions of plan candidates the plan with the least estimated cost. However, traditional optimization techniques cannot be readily integrated into systems that aim to support novel data analysis use cases. For example, the use of user-defined functions (UDFs) can significantly limit the optimization potential of data analysis programs. Furthermore, lack of detailed data statistics is common when large amounts of unstructured data is analyzed. This leads to imprecise optimizer cost estimates, which can cause sub-optimal plan choices. In this thesis we address three challenges that arise in the context of specifying and optimizing data analysis programs.

First, we propose a parallel programming model with declarative properties to specify data analysis tasks as data flow programs. In this model, data processing operators are composed of a system-provided second-order function and a user-defined first-order function. A cost-based optimizer compiles data flow programs specified in this abstraction into parallel data flows. The optimizer borrows techniques from relational optimizers and ports them to the domain of general-purpose parallel programming models.

Second, we propose an approach to enhance the optimization of data flow programs that include UDF operators with unknown semantics. We identify operator properties and conditions to reorder neighboring UDF operators without changing the semantics of the program. We show how to automatically extract these properties from UDF operators by leveraging static code analysis techniques. Our approach is able to emulate relational optimizations such as filter and join reordering and holistic aggregation push-down while not being limited to relational operators.

Finally, we analyze the impact of changing execution conditions such as varying predicate selectivities and memory budgets on the performance of relational query plans. We identify plan patterns that cause significantly varying execution performance for changing execution conditions. Plans that include such risky patterns are prone to cause problems in presence of imprecise optimizer estimates. Based on our findings, we introduce an approach to avoid risky plan choices. Moreover, we present a method to assess the risk of a query execution plan using a machine-learned prediction model. Experiments show that the prediction model outperforms risk predictions which are computed from optimizer estimates.

Zusammenfassung

In der Vergangenheit wurde die überwiegende Mehrheit der Datenanalyseanwendungen durch die Aggregation von relationalen Daten abgedeckt. Seit einigen Jahren entwickelt sich ein Trend der "Big Data" genannt wird und der große Auswirkungen auf den Bereich der Datenanalyse hat. Im Vergleich zu bisherigen Analyseanwendungen, werden nun wesentlich größere Datenmengen mit deutlich aufwändigeren und vielfältigeren Analysemethoden wie zum Beispiel Techniken der Informationsextraktion, des Data Minings, und Verfahren des maschinellen Lernens ausgewertet. Dabei werden auch Daten in die Analyse einbezogen, die weniger stark oder überhaupt nicht strukturiert sind. Die Veränderungen der Eigenschaften von Datenanalyseanwendungen wirken sich auch auf die Anforderungen an Systeme zur Datenverarbeitung aus. Aufgrund des gestiegenen Datenvolumens und des durch komplexere Analyseverfahren deutlich höheren Berechnungsaufwands müssen Daten massiv parallel verarbeitet werden. Die gestiegene Vielfalt von Analyseverfahren und die geringere Struktur der Daten erfordern häufig den Einsatz von benutzer-definierten Funktionen und Datenstrukturen. Viele traditionelle Datenbanksysteme sind nicht flexibel genug, um diesen Anforderungen gerecht zu werden. Deshalb gibt es ein großes Interesse an neuen Programmierabstraktionen mit denen komplexe und parallele Datenanalyseanwendungen spezifiziert und effizient ausgeführt werden können.

Der Erfolg der Anfragesprache SQL hat die Vorzüge von deklarativer Anfragespezifikation, wie zum Beispiel Optimierungspotenzial und Benutzerfreundlichkeit, deutlich gezeigt. Heute nutzt nahezu jedes relationale Datenbanksystem einen Anfrageoptimierer der deklarative Anfragen in physische Ausführungspläne übersetzt. Kosten-basierte Optimierer sind in der Lage aus Milliarden von möglichen Plänen einen effizienten Plan auszuwählen. Allerdings lassen sich traditionelle Optimierungsmethoden nicht ohne weiteres in Systeme integrieren, die neuartige Anwendungsfälle von Datenanalyse unterstützen wollen. Zum Beispiel kann der Einsatz von benutzer-definierten Operationen das Optimierungspotenzial sehr stark reduzieren. Darüberhinaus sind selten detaillierte Datenstatistiken verfügbar, wenn große unstrukturierte Datensätze analysiert werden. Fehlende Statistiken haben häufig ungenaue Kostenschätzungen des Optimierers und somit die Auswahl von suboptimalen Ausführungsplänen zur Folge. In dieser Arbeit adressieren wir drei Herausforderungen im Kontext der Spezifikation und Optimierung von parallelen Datenanalyseprogrammen mit benutzer-definierten Funktionen.

Zunächst stellen wir ein paralleles Programmiermodell mit deklarativen Eigenschaften vor um Datenanalyseprogramme als Datenflußprogramme zu spezifizieren. In diesem Modell bestehen Datenverarbeitungsoperatoren aus einer system-eigenen Funktion zweiter Ordnung und einer benutzer-definierten Funktion erster Ordnung. Ein kosten-basierter Optimierer übersetzt Datenflußprogramme, die in unserem Programmiermodell definiert wurden, in parallele Datenflüße. Unser Optimierer baut auf viele Techniken der relationalen Optimierung auf und überträgt sie in die Domäne von universellen parallelen Programmiermodellen.

Zweitens präsentieren wir einen Ansatz zur Verbesserung der Optimierung von Datenflußprogrammen, die benutzer-definierte Operatoren mit unbekannter Semantik enthalten. Wir identifizieren Eigenschaften von Operatoren und Bedingungen, um die Reihenfolge von benachbarten benutzer-definierten Operatoren zu verändern ohne die Semantik eines Programms zu ändern. Wir zeigen wie diese Eigenschaften für benutzer-definierte Operatoren vollautomatisch mit Hilfe von statischer Codeanalyse aus deren Quellcode extrahiert werden können. Mit unserem Ansatz können viele relational Optimierungen wie zum Beispiel die Optimierung der Reihenfolge von Filtern, Joins und Aggregationen emuliert werden ohne jedoch auf relationale Operatoren beschränkt zu sein.

Drittens analysieren wir den Einfluß von sich verändernden Ausführungsbedingungen wie zum Beispiel variierenden Prädikatselektivitäten und verfügbaren Hauptspeichermengen auf die Laufzeit von relationalen Ausführungsplänen. Wir identifizieren Planeigenschaften, die deutliche Laufzeitschwankungen auslösen können. Im Fall von ungenauen Optimiererschätzungen können Pläne, die diese Eigenschaften enthalten, ein sehr großes Risiko darstellen. Wir präsentieren einen Ansatz, um die Auswahl von riskanten Plänen zu vermeiden. Darüberhinaus stellen wir eine Methode vor, um das Risiko von Ausführungsplänen mit Hilfe eines maschinell-gelernten Modells vorher zusagen. Unsere Evaluation zeigt, dass mit unserem Vorhersagemodell das Risikopotenzial eines Plans besser abgeschätzt werden kann als mit Hilfe eines kosten-basierten Optimierers.

Contents

1 Introduction

Contents

1.1 Motivation

"Big Data" is a topic that is popular nowadays and raises many expectations [145]. As with most trends, there is no consensus of what Big Data actually means and there are several perspectives on it, including ecological, social, legal, and of course technological views. In a nutshell, Big Data describes the idea to extract valuable information from very large and diverse data sets. This is similar to the mission statement of data mining. What makes a difference though, is the size, diversity, and origin of the analyzed data and the context in which the analysis is performed. The amount of available data in many domains is exploding due to excessive collection of machine-generated, sensor-collected, and user-generated data [20, 24, 45, 46, 65, 188]. One aspect of the Big Data trend is to combine data sets from such diverse sources to extract previously unknown information. Often, Big Data projects aim to optimize existing processes, give guidance for complex decisions, or enable new services. Given the relevant data sets with the sufficient size and quality and the right data analysis tools, significant improvements are possible. For example, the efficiency of logistics can be improved by analyzing GPS traces of trucks, adapting their routes, and reducing their fuel consumption [157]. Other use cases include predicting shortage of supplies, optimizing manufacturing processes, utilization of shared resources, and analyzing scientific data from astronomy, biosciences, or geosciences [20, 24, 45]. The ability to quickly analyze vast amounts of data will empower future scientific breakthroughs and will be a significant advantage of enterprises over competitors that lack this capability.

From a technological point of view, Big Data challenges traditional data management in several ways. Most obviously, the amount of data to be analyzed is constantly growing at high rates. Data sets are produced by humans (e. g., social media interactions, clickstreams, and open data collections such as Wikipedia or OpenStreetMap), collected by machines and sensors (e. g., network monitoring, RFID tracking, traffic monitoring, and GPS position recording), or generated from scientific experiments (e. g., particle research, genome sequencing, climate simulations) [1, 7]. At the same time the cost for storing data is decreasing, making it affordable to store large amounts of data to analyze it later. Also the characteristics of the data that is analyzed has changed. While in the past, data analysis was mostly performed on structured data sets, new use cases include the processing of graph-structured, semi-structured, and unstructured data, such as text or image data [46, 148]. However, not only the data which analyzed changes. Also the techniques and methods to analyze this data have become more complex and demanding. While in the past data was commonly analyzed using relational queries, today information extraction, text mining, data mining, and machine learning techniques and algorithms are commonly applied. These methods are for example used to extract structured information from text, cluster data points, or compute predictions or classification models [63, 148, 152]. Due to the variety of data formats and the diversity of analytical tasks, data analysis applications often cannot be realized by exclusively using domain-specific languages but commonly require some kind of custom implementations.

Big Data analytics pose a number of requirements on systems that aim to support these kinds of workloads. First of all, Big Data processing systems need to be able to run complex analytical tasks on large amounts of data. This demands for large capacities of I/O and compute resources. A common approach to address this requirement is distributed and parallel processing. As many data sets are expected to grow over time, data processing systems need to be able to flexibly scale with increasing data set sizes. In addition to parallel execution, data processing systems need to focus on efficiency to reduce the demand for compute resources and consequently for power and cooling. Ease of use is another important aspect for data processing systems and places requirements on the abstractions to define data processing tasks. A good abstraction hides unnecessary complexity from users, but is expressive enough to enable all relevant use cases. Since parallel computing is challenging and requires excellent programming skills to deal with communication, concurrency and fault tolerance, abstractions to hide these complexities have become important. Declarative programming models are a natural fit for such abstractions. When compiling a declarative program into an executable program, a compiler often can choose from several equivalent execution strategies with different execution performance. Hence, the compilation task is an optimization problem to identify the most efficient execution strategy. Declarative specification eases program development, since the difficult choice of an efficient execution strategy is left to the system's compiler [97, 186]. This improves developer productivity and also lowers the requirements on the skill set of developers. Moreover, program optimization decouples the program specification from the data to process and the execution environment because an optimizer can choose the best plan depending on the current situation. In

fact, the declarative query language SQL with the underlying relational algebra and the sophisticated query optimizers are among the reasons for the success of relational database systems. While SQL is a good abstraction for traditional analytical tasks, today's use cases demand for custom data types and user-defined functions which must be supported as first class citizens by programming abstractions of Big Data processing systems.

Until a few years ago, parallel relational database systems (RDBMS) have been the predominant solution to process large amount of data. However, these systems do not address all requirements discussed before. Parallel RDBMSs are tailored towards the relational data and processing model. Tasks that fall into this domain can be efficiently processed. Relational database systems offer a declarative programming model (SQL) and have sophisticated query optimizers that compile complex queries into efficient execution plans. Hence, users with little background on query processing can state complex queries which can be efficiently executed. However, tasks that process unstructured data or require complex custom code are not so easy to realize, if at all [65]. In addition, parallel RDBMSs are typically rather costly solutions. This is due to high licensing costs, which often depend on the size of the data sets or amount of compute resources used by the database. Such licensing terms can cause high costs for systems that are intended to store and process large amount of data. Furthermore, parallel database systems often require special hardware setups, are hard to scale-out, and need to be maintained by skilled database administrators [14].

MapReduce [64] and distributed file systems [92] have been designed to overcome some of the limitations of parallel RDBMSs. Distributed file systems scale to thousands of commodity hardware servers, reliably store large amounts of unstructured data, and offer high I/O performance. The MapReduce programming model supports custom data types and user-defined functions and hides the complexity of parallel programming from the user, such as network communication, concurrent programming, and fault tolerance. However, MapReduce's design focuses more on scalability and fault-tolerance than on resource and programmer efficiency. MapReduce is a low-level programming model that requires imperative programming and does not offer declarative task specification and optimization. While relatively simple processing tasks can be easily implemented, complex data analysis jobs are cumbersome to express in MapReduce and require comprehensive system insight and programming experience. This deficiency led to the design and implementation of higher-level languages and programming models on top of MapReduce such as Pig [169], Hive [198], Jaql [28], and Cascading [44]. These languages and programming models offer more declarative programming abstractions that include primitives such as joins, filters, and aggregations and compile queries or programs into MapReduce programs. Hence, they significantly ease the implementation of complex analysis tasks which are executed in a massively parallel and fault-tolerant fashion using MapReduce. Essentially, Pig, Hive, Jaql, and Cascading leverage MapReduce as an execution engine similar to how relational database systems use their engines to execute SQL queries. However, using MapReduce as an execution engine comes at cost of processing efficiency compared to the execution engines of relational database systems [173, 194].

Given the shortcomings of current systems to define and execute complex analytical tasks on large amounts of data, there is a need for a system that provides declarative task specification, optimization, and efficient parallel execution.

1.2 Scope of Thesis

This thesis focuses on challenges and problems related to the specification and optimization of parallel data flow programs. In this section, we define the scope of our contributions and briefly discuss their context.

1.2.1 Abstractions for Parallel Data Analysis Tasks

The success of relational database systems is, among other reasons, due to the powerful combination of a declarative query language (SQL) and sophisticated query optimization and compilation technology. When stating queries, users describe the desired result without defining how the result should be computed. This separation of logical task description and physical execution also allows for parallel execution of SQL queries. Due to its clean abstraction and ease of use, SQL has become the de-facto standard to interact with database systems. However, SQL builds on a small set of relational operators and was designed for well-structured relational data. Although, most database systems offer interfaces for user-defined functions, these are often too restrictive and hard to specify. With the growing number of Big Data analysis use cases, data analysis tasks commonly include heterogeneous, non-relational data and cannot be easily specified with SQL [63, 148, 152].

As a consequence, alternative programming models have been defined to cope with the new challenges. The most prominent representative of Big Data programming abstractions is MapReduce [64]. MapReduce offers two parallelization primitives, Map and Reduce, which process user-defined data types by calling user-defined functions. These parallelization primitives and MapReduce's execution model enable parallel execution of MapReduce programs and allow the developer to focus on application logic instead of issues related to parallel programming which are transparently handled by MapReduce. The templates of the user-defined functions and data types are generic such that a wide variety of data processing tasks can be realized using the MapReduce programming model. MapReduce's execution model is fixed, i. e. each MapReduce job is executed exactly the same way. By combining user-defined processing and automatic parallelization, MapReduce addresses several needs of Big Data processing and has become very popular. However, some analysis tasks cannot be easily defined using MapReduce and need to be forced into the programming model [65]. Moreover, often implicit knowledge about the fixed execution model is required to define tasks or to enable more efficient execution. This clearly contradicts with the motivation of declarative task specification. Higher-level languages such

as Pig [169], Hive [198], Jaql [28], and Cascading [44] have been designed to overcome this limitation of MapReduce. Queries or programs specified in these higher-level abstractions are compiled into MapReduce programs. Hence, they offer a convenient interface to specify parallel data analysis tasks. However, higher-level programming abstractions on MapReduce suffer from MapReduce's fixed execution model which can lead to inefficient execution plans with high resource consumption especially for more complex analysis tasks.

Neither SQL, MapReduce, nor higher-level languages on MapReduce fulfill the requirements for a Big Data programming model. While SQL offers declarative query specification and efficient execution, it lacks extensive support for user-defined data types and functions. On the other hand, MapReduce is built around user-defined data types and functions, but does not offer a declarative task specification. Higher-level languages and programming models such as Pig, Hive, or Cascading offer declarative abstractions, user-defined functions, and massively parallel execution. However, they are limited by inefficiencies of MapReduce execution model.

1.2.2 Optimizing Data Flows with UDF Operators

Query optimization is a key technology to improve the performance and efficiency of data processing systems [12, 18, 89, 101, 186, 187, 213]. In relational DBMSs, the choice of the query execution plan can result in a difference in execution time of multiple orders of magnitude [83, 186, 191]. Especially the order of operators can have a significant impact on the performance of an execution plan. For large-scale systems that concurrently execute multiple analysis jobs and process large amounts of data, suboptimal plan choices can waste valuable compute resources, such as I/O operations and CPU cycles, and significantly impact the overall performance of a system.

The characteristics of Big Data applications include heterogeneous data and complex analytical tasks such that data analysis programs often include user-defined functions. However, user-defined functions pose a big challenge for data flow optimization because their semantics are not known by the processing system. Optimizing queries with UDFs has been extensively researched in the context of relational DBMS [53, 118, 119]. However, the UDF interfaces that are offered by today's MapReduce-inspired systems are more generic compared to those offered by RDBMS because they have less well-defined semantics and operate on data without explicit schema information [64]. There have been approaches to optimize the execution of MapReduce programs by tuning the parameters of the MapReduce execution engine [122] or replacing file scans with index scans [42]. We are not aware of any optimization methods to improve the order of operators with MapReduce-style user-defined functions.

1.2.3 Assessing the Risk of Relational Data Flows

The execution time of a data flow program depends on the program itself, the data to process, and the execution environment. Cost-based data flow optimizers rely on information about the query, the data, and the execution environment that is available at optimization time to estimate the cost of execution plans and pick the one with the least estimated costs. The accuracy of the available information is important for the effectiveness of this approach. Missing or inaccurate information can result in choosing execution plans that consume more compute resources and have significantly worse performance than other plans [131, 158, 164, 176].

Lack of information at optimization time is a major challenge in cost-based optimization [17, 99, 102, 159]. Compared to relational database systems, this problem is even bigger in the context of Big Data analysis systems. Traditional DBMSs collect basic statistics such as table cardinality when data is loaded and more detailed statistics as for example attribute histograms when requested by the user or automatically at runtime [130, 135, 176]. In contrast, detailed statistical information about the data is often not available in today's data analysis environments because ad-hoc data is read from distributed file systems or processed by user-defined functions with unknown semantics at runtime. Furthermore, processing tasks contain UDFs with unknown complexity and are executed on large, possibly virtualized, clusters.

An important observation is that not all plans are equally sensitive to lacking information [17, 60, 99, 100, 204]. While some plans heavily rely on incorrect assumptions of the optimizer due to missing information and exceed the expected execution time by orders of magnitude, other plans might tolerate faulty assumptions with little impact on their performance. Especially plans which "aggressively" aim to leverage assumed data characteristics such as small cardinalities of intermediate results can easily degrade.

Assessing and reasoning about the sensitivity of execution plans with respect to imprecise optimizer information is a hard and not extensively researched topic.

1.3 Contributions of Thesis and Impact

This thesis discusses approaches to specify, automatically optimize, and efficiently execute parallel data processing tasks. It also addresses the problem of suboptimal optimizer plan choices due to inaccurate optimizer estimates. We present our contributions in detail in Sections 3, 4, and 5 and summarize them in the following.

In Chapter 3, we propose the PACT programming model to define parallel data flows. PACT is a generalization of the MapReduce programming model with declarative properties. It models programs as directed acyclic graphs and features parallelizable user-defined functions and

custom data types. We develop methods to optimize PACT programs, which are based on techniques from relational query optimization, such as cost-based optimization and interesting property reasoning. While the PACT programming model is similar to other high-level programming abstractions such as Cascading [44] or Pig [169], the optimization differs because programs are not translated into MapReduce jobs but into data flows which are similar to execution plans of relational database systems. We show that the PACT programming model considerably eases the definition of advanced data analysis tasks and significantly improves their performance compared to MapReduce implementations. This work is a step towards closing the gap between declarative and efficient RDBMS and massively parallel and expressive MapReduce-based systems. PACT is a core component of Stratosphere [10], a system for massively parallel data analytics. The contributions of Chapter 3 are joint work with Stephan Ewen.

In Chapter 4, we develop optimization techniques to reorder user-defined operators in data flows. We present and prove sufficient conditions to reorder two successive UDF operators of PACT programs. Our conditions evaluate certain properties of the user-defined functions and are applicable to many systems that support MapReduce-style UDFs. We leverage static code analysis techniques to automatically extract the required properties from UDFs and design an algorithm to enumerate equivalent data flows by swapping subsequent operators. The implementation of our optimization techniques is based on the Stratosphere system. The evaluation shows that our approach is able to perform important optimizations known from relational query optimization including filter and projection push-down, join reordering, and holistic aggregation push-down, while not being limited to relational operators. Some contributions of Chapter 4 are joint work with co-authors of the corresponding publication [129]. The conditions for reordering UDF operators were defined and proven in collaboration with Mathias Peters and Matthias Sax. Astrid Rheinländer and Matthias Sax contributed to the evaluation of the approach.

In Chapter 5, we perform an extensive experimental study to analyze the impact of inaccurate estimates on the quality of optimizer plan choices. For this study we execute 306 different execution plans for 14 analytical queries under varying execution conditions and measure their execution time. We analyze the resulting data and identify operators and plan patterns that are sensitive to changing execution conditions. We find that predicting the sensitivity of an execution plan is not possible by looking at individual operators but requires a holistic view at the plan. We propose a risk score to assess the sensitivity of query execution plans with respect to varying execution conditions and show how it can be used to avoid risky plan choices. Using the data obtained from the experimental study, we develop a method to predict the risk score of query execution plans based on a machine-learned regression model. The evaluation shows that our machine learning approach outperforms predictions based on optimizer estimates. To the best of our knowledge, this work is the most extensive study to analyze plan robustness and the first approach to predict the robustness of query execution plans using machine learning techniques.

Parts of this thesis have been published as follows:

- Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *SoCC '10: Proceedings of the ACM Symposium on Cloud Computing 2010*, pages 119–130, New York, NY, USA, 2010. ACM.
- Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively parallel data analysis with PACTs on Nephele. In *PVLDB*, 3(2):1625–1628, 2010.
- Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. MapReduce and PACT - Comparing data parallel programming models. In *BTW*, pages 25–44, 2011.
- Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. In *PVLDB*, 5(11):1256–1267, 2012.
- Fabian Hueske, Aljoscha Krettek, and Kostas Tzoumas. Enabling operator reordering in data flow programs through static code analysis. In *XLDI Workshop, affiliated with ICFP*, 2012.
- Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. Peeking into the optimization of data flow programs with MapReduce-style UDFs. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1292–1295, 2013.
- Fabian Hueske and Volker Markl. Optimization of massively parallel data flows. In *Large-Scale Data Analytics*, pages 41–74. Springer New York, 2014.
- Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, pages 1–26, 2014.

The results of this thesis have been adopted by the research community and are published as open source software. The PACT programming model [23] is a core component of the Stratosphere system [10] and serves as a basis for several publications [34, 43, 80, 117, 139, 140, 149, 180, 182].

The Apache Flink project [84] is a fork of the Stratosphere system and evolved into a top-level project of the Apache Software Foundation in 2014. By now, Apache Flink is a very active open source project with a steadily growing community and experiences lots of interest. Several

companies are using Apache Flink in production. Multiple open source projects integrate with Apache Flink and use it as an execution engine, such as Apache Mahout [156], Cascading [44], and Apache MRQL [165].

1.4 Outline of Thesis

In the following we give the outline of this thesis. Chapter 2 discusses the background and related work for parallel data processing and optimization of parallel data flows. In Chapter 3 we present our abstraction to specify parallel data processing tasks. Chapter 4 discusses the optimization of data flows that embed MapReduce-style user-defined functions. Chapter 5 evaluates the sensitivity of execution plans with respect to incorrect assumptions at optimization time and presents an approach to avoid too risky execution plans. Finally, Chapter 6 concludes this thesis and gives an outlook.

2 Background

Contents

Analyzing more data than a single state-of-the-art computer is capable to store or to process of is not a new requirement. In fact, the design of the first parallel relational database systems started in the late 1970's [67]. Since then, several commercial parallel database systems have been built and constantly improved [21, 22, 32, 110, 187]. The publication of Google's MapReduce framework in 2004 [64] accelerated the evolution of parallel data processing systems and triggered several new system designs.

In this chapter, we present the foundations and the state-of-the-art of parallel data processing and data flow optimization. Section 2.1 introduces important concepts of parallel data processing and presents different types of systems including parallel relational database systems, MapReduce-based systems, and parallel data flow systems. Section 2.2 discusses the state-of-the-art in the area of optimizing abstractions for parallel data analysis such as relational queries, MapReduce programs, and higher-level programming abstractions. Finally, Section 2.3 summarizes this chapter.

2.1 Parallel Data Flow Processors

This section discusses parallel data processing systems. We briefly introduce the basic principles of parallel data processing and continue to present the evolution of parallel data processing systems. We discuss three system families, namely parallel relational database systems, MapReduce-based systems, and systems that execute generic parallel data flows. Finally, we compare the advantages and shortcomings of these system families.

2.1.1 Principles of Parallel Data Processing

Processing data in parallel on multiple machines is not a new trend. At any point in time, there have been applications that challenged the compute resources which were available in a single state-of-the-art computer. Depending on the use case, CPU performance, main memory size, storage capacity, or I/O bandwidth limit the performance of data analysis programs. One approach to overcome this limitation is to split the computation and let multiple machines collaboratively compute the desired result. By distributing computations to multiple machines, their resources can be jointly utilized, resulting in improved performance. In order to enable parallel data processing, some basic challenges need to be addressed. First, a data processing program must be split into multiple individual computation tasks that can be distributed to machines. Second, processes that participate in a parallel computation need a mechanism to communicate with each other in order to exchange data and synchronize their operations. In this section, we briefly discuss these two principled aspects of parallel data processing[1].

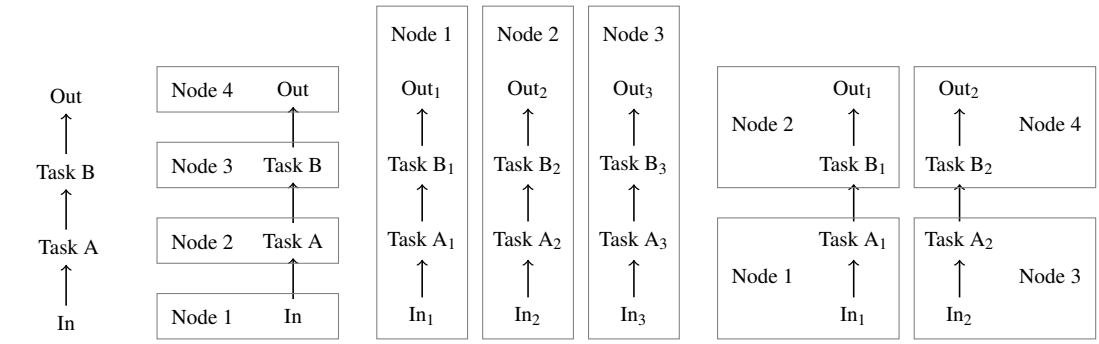
Types of Parallelism There are basically two ways to split a data processing program into individual parts that can be processed by different machines, *task parallelism* and *data parallelism* [68].

Data processing programs usually consist of several processing steps or operations. Figure 2.1(a) shows a program consisting of four steps, read input data (In), apply operation A (Task A), apply operation B (Task B), and writing the result (Out). In task parallelism, a data processing program is split along its processing steps into program fragments that are sent for execution to different machines. Figure 2.1(b) shows how the example program is split into four parts that are processed by four machines (Node 1 to 4). This distribution of tasks to different machines requires that the output of an operation must be transferred to the machine that applies the next operation. In this setting, each machine applies its operation to the full data set, i. e., all data items are routed through each machine.

[1] Further important aspects include load balancing, fault-tolerance, and task scheduling. These aspects are more application specific and discussed later in the context of individual systems.

In contrast, data parallelism does not split the program into program fragments, but partitions the data to process. Figure 2.1(c) shows how the computation of the example program can be parallelized and distributed to three machines using data parallelism. Each machine applies all operations of the program to a subset of the overall data. Whether data needs to be exchanged between processing machines depends on the characteristics of the program.

Comparing both types of parallelism, the maximum degree of parallelism for task parallelism is limited by the complexity of the program, i. e., the number of operations. In contrast, data parallelism is limited by the number of data items. Regarding load balancing, task parallelism can suffer from tasks with different computational complexity, while data parallelism has to cope with skewed data distributions. An important aspect of task parallelism are blocking operations that consume the entire data set before they emit their result, such as a sort operation. In a sequence of tasks, such blocking operations split a parallel program into pipelines of operations that concurrently processes data. In practice, many analytical data processing systems use a combination of task and data parallelism as shown in Figure 2.1(d). Programs are split up into groups of (pipelined) operators, each of which processing a subset of the overall data set.

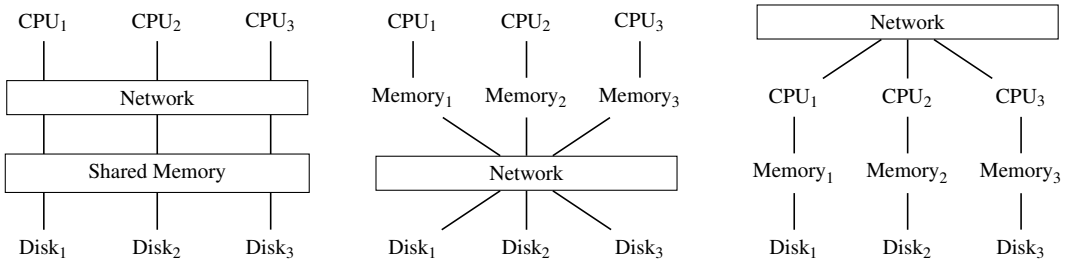


(a) Program (b) Task Parallelism (c) Data Parallelism (d) Hybrid Parallelism

Figure 2.1: Types of parallelism

Process Communication Processes that collaboratively execute a data analysis task and run on multiple machines need to communicate with each other. This communication includes exchanging data and synchronizing their operations. There are three alternative architectural system designs that enable inter-machine communication, namely *shared-memory*, *shared-disk*, and *shared-nothing* [68, 171]. Figure 2.2 depicts these designs.

The shared-memory architecture enables machines to access the memory of remote machines (Figure 2.2(a)) [26]. This allows machines to communicate by reading from and writing to memory. A problem with this design are the differences in access latency and throughput, as well as coordination of concurrent memory accesses (locking). In the shared-disk design [178], all machines access a shared persistent data store, such as a storage area network (SAN) as



(a) Shared Memory (b) Shared Disk (c) Shared Nothing

Figure 2.2: System architectures for parallel data processing systems (c. f. [68])

shown in Figure 2.2(b). Similar to the shared-memory approach, concurrent data access must be coordinated. Both approaches have limitations in setups that require massive scale-out to a high number of machines. This is mainly due to the effort of coordinating a large number of concurrent accesses to a shared resource. The predominant architectural design for parallel data processing systems is the shared-nothing architecture (Figure 2.2(c)) [68]. In this architecture, each machine accesses only its local disk storage and main memory and exclusively communicates through network messages with other machines. Concurrent data accesses must only be coordinated among processes which locally run on a machine.

Compute-intensive applications such as high performance computing (HPC) applications [61] and specialized data analysis solutions, which can avoid (or significantly reduce) the overhead of distributed locking, often chose shared-memory architectures [155]. However, systems with a focus on large-scale general-purpose data analysis applications are usually based on shared-nothing architectures [193].

2.1.2 Parallel Relational Database Systems

Parallel relational database systems (RDBMS) are the first generation of systems for analyzing large amounts of data. A parallel RDBMS provides a single-instance interface similar to a stand-alone database system. However, the system is running on multiple machines that jointly store and query data. The origins of parallel DBMS date back to the late 1970's. Teradata and Tandem [78, 210] were among the first commercial products pioneering parallel system architectures for database systems. First research prototypes following this design were Gamma [67] and Bubba [35]. DeWitt and Gray [68] give an overview of the state-of-art in 1990 and discuss the advantages of parallel database systems over the database machine approach which relied on specialized instead of commodity hardware. Since then, all major database vendors have added a parallel database management system to their portfolio. Starting in the early 2000's, a new

generation of parallel database systems was designed which is focused on large-scale analytical workloads. Among these systems are Greenplum [201], Aster Data [87], and Netezza [62]. These systems are designed for massive parallelization with advanced fault tolerance mechanisms and partially add support for more generic programming models than SQL.

In this section, we give a brief overview of the technology behind parallel RDBMSs. We focus on technology and database systems for analytical workloads and do not include database systems and techniques for transactional workloads. We discuss the system architecture of parallel RDBMSs, data storage techniques, and explain how queries are processed by these systems.

Data Storage in Parallel RDBMS Most parallel database systems follow the shared-nothing system architecture, where each machine uses its local storage and memory for data processing. The data which is stored in such a system needs to be distributed among all machines to be processed in parallel [68, 143, 161]. The relational data model is a good fit for this requirement. Relations can be easily split into horizontal partitions, i. e., each record of a relation is assigned to exactly one partition. Each partition is assigned to one or multiple machines of the system. The assignment of a record to a partition can be random, round-robin, or based on the value of an attribute or a combination of attributes (range- or hash-partitioning). Depending on the chosen partitioning method, the amount of data that needs to be transferred over the network to compute the result of a query can vary by large factors. Therefore, the choice of partitioning methods can have a significant impact on the performance of a system and should be chosen with respect to the expected workload. For smaller relations it can be beneficial to fully replicate them to each machine of the system and in order to avoid the shipping of much larger relations.

Query Processing in Parallel RDBMS SQL is a declarative query language. In contrast to imperative programs, SQL queries describe the requested result instead of exactly specifying the way to compute it. In order to evaluate a SQL query, a query compiler translates it into an execution plan, which is executed. Due to the independence of query specification and execution and the fact that relational expressions can be easily parallelized, SQL queries can be run on stand-alone and parallel database systems without any modifications.

The execution plan of a SQL query is a data flow built from operators, where data flows from one operator to another operator. Optimizing SQL queries for parallel execution builds on query optimization techniques for stand-alone settings and extends these. An important characteristic of optimization in both settings is pipelined execution. Pipelining has the goal to perform as much processing as possible "on-the-fly" without the need to materialize data, i. e., write it to disk. The most important difference to optimization in a non-parallel setting is the consideration of data shipping costs, which can easily dominate the cost of query execution. Hence, query optimizers for parallel execution are guided by their cost models to reduce the amount of shipped

data as much as possible by maximizing local processing on each individual machine. An important technique is to leverage data locality through well-partitioned relations [143]. While filter and projection operators can be locally executed on any partitioning, aggregations and joins require the system to organize data by their grouping or join attributes. An aggregation can be locally executed if the data is hash- or range-partitioned on a subset of its grouping attributes. An equi-join can be performed without communication if either both involved relations are partitioned on their join attributes (partitioned join) or one of both relations is fully replicated and locally accessible at each partition of the other relation (broadcast join). Further techniques to reduce the amount of shipped data include specialized physical operators, such as semi-joins or partial aggregates [54, 143]. While these techniques can considerably reduce the need for data shipping, complex queries can require to repartition the data once or multiple times. Finally, all partial results need to be collected in order to return the result to the client. A major challenge for parallel database systems is data skew. Non-uniform data distributions can significantly impact the performance of a query, since the execution time of a query depends on the completion of last parallel process [147].

2.1.3 MapReduce Systems

MapReduce systems evolved as the second generation of systems to analyze large amounts of data. In 2003 and 2004, Google presented its framework for massively parallel data processing consisting of the distributed *Google File System* (GFS) [92] and the processing framework *MapReduce* [64, 65]. Google's applications for MapReduce included processing of web crawls, search index creation, and log file analysis. These kinds of analytical and data processing tasks did not fit the predominant relational data and processing model well. Moreover, Google required a system that was able to cost-efficiently and reliably scale to thousands of machines.

Shortly after the publication of MapReduce, the Apache Hadoop [109] project started to implement a distributed file system and a MapReduce platform. The free availability of a MapReduce system and its source code lead to numerous efforts in research, industry, and the open source community. Today, Hadoop has become a standard for processing large amounts of data.

In this section, we introduce the basic concepts of the Google File System and MapReduce's programming and execution models. We also give a brief overview of different research and open source efforts that were triggered by the publication of MapReduce focusing on aspects that are relevant in the context of this thesis, i. e., specification and optimization data processing tasks.

Google File System The Google File System is a distributed file system that was specifically designed to run on large clusters of unreliable commodity hardware and to support large-scale

data processing tasks [92]. Due to these requirements, it differs from previous designs for distributed file systems.

GFS follows a master-slave architecture with a centralized GFS master and multiple GFS chunk servers. The GFS master is responsible for storing metadata, such as the file system directory structure, file storage locations, and access permissions. GFS chunk servers store the actual data which is written to and read from the file system. Clients connect to the master to look up file locations at the GFS master and coordinate reading and writing directly with the chunk servers.

GFS was optimized to serve large files of multiple gigabytes. Files are split into fixed-sized chunks. These chunks are distributed and replicated over all chunk servers of the file system. A chunk server stores its chunks on its local file systems. The GFS paper [92] mentions a default chunk size of 64 megabytes and a default replication of three. This design has several benefits. First, a file can be read in parallel by concurrently reading multiple of its chunks from different chunk servers. Thereby, the I/O bandwidth of all involved hard disks and servers is combined. Second, replication ensures fault-tolerance. In case some machines or disks fail, other replicas of the lost chunks are available and can be used to restore the replication level. However, GFS does not perform well for random access read and write operations. The open source project Apache Hadoop [109], implements its Hadoop distributed file system (HDFS) after the design blueprint of Google's GFS paper.

MapReduce Programming Model The MapReduce programming model is centered around two parallelization primitives, *Map* and *Reduce*. Map and Reduce are second-order functions. A data processing task is specified as two user-defined first-order functions (UDFs), one for Map and one for Reduce. These UDFs are executed in parallel on different subsets of the processed data set. Hence, MapReduce leverages the principle of data parallelism to scale-out data processing, similar to parallel RDBMSs. In addition, MapReduce uses task parallelism to distribute the computation of Map and Reduce functions.

MapReduce's data model differs from the relational model. A data set is represented as a bag of key-value pairs. Each pair represents a data item and consists of a key and a value element. Both, key and value can be of user-defined types. The only requirement for key and value types is serializabilty because the processing system needs to be able to write data to and read data from hard disk and network connections. In addition, key types must also provide methods for comparison and hashing. Hence, keys and values can be atomic data types, such as Integers or Strings, but also more complex data structures, such as annotated text documents or objects representing geographic locations. Compared to relational database systems, MapReduce's data model is more flexible and enables the processing of complex, heterogeneous data sets.

In addition to user-defined data types, a data processing task also consists of two user-defined functions, a Map UDF and a Reduce UDF, which are called by the corresponding second-order

parallelization function, Map or Reduce. The function signatures of both UDFs are shown in the following.

$$\texttt{mapUDF}\ (k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$\texttt{reduceUDF}\ (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

A MapReduce program is works as follows. First, the input of the program, a bag of key-value pairs of type (k_1, v_1), is processed by the Map function. The function calls its UDF exactly once for each key-value pair (k_1, v_1) of its input bag. The Map UDF may produce arbitrary many key-value pairs of a possibly new type (k_2, v_2), i. e., neither the key nor the value type needs to be preserved. Subsequently, the Reduce function processes the output of the Map function and calls its UDF exactly once for each distinct key of Map's output data set. With each call, Reduce provides the distinct key of type k_2 and a list of all values of type v_2 which are associated with the distinct key to its UDF. The Reduce UDF may as well produce an arbitrary number of key-value pairs of a new type (k_3, v_3). All UDF invocations are independent from each other, which is a crucial requirement for data parallelism. It is important to note that the parallelization primitives Map and Reduce do not exactly correspond to their counterparts, which are known for functional programming languages. Due to the focus on user-defined functions and data types, MapReduce is a versatile programming model and many data processing tasks can be intuitively expressed this way.

The MapReduce programming model is also flexible in its choice of data sources. A MapReduce program requires an input format that provides the key-value pairs, which are processed by the Map function. The programming model does not restrict the source from which data can be read or the method to create pairs. Hence, input formats can read data from any data source, such as a file system, a relational database, a distributed hash-table, or any other data source. However in order to enable parallel execution, the input format needs to be able to read different subsets or partitions of a data set in parallel. Each subset or partition of the input data is called an input split. An input format needs to provide a list of input splits for its input. Finally, a MapReduce program requires an output format to emit the result of a MapReduce program. The output format defines the data store and format of a program's output.

The performance of a MapReduce program depends to a large extent on the amount of data that is transferred over the network in order to group the data by key as required by the Reduce function. The MapReduce programming model offers an additional optional Combine function to reduce the data that needs to be shipped over the network. The Combine function is applied after the Map function and partially aggregates data that is local on a machine by the Reduce key. Whether or not a Combine function can be used depends on the characteristics of the Reduce function, i. e., the Reduce function needs to be commutative and associative. The signature of the Combine function is

$$\texttt{combineUDF}\ (k_2, [v_2]) \rightarrow [(k_2, v_2)].$$

The Combine function must preserve its input types and must not modify the key. Usually, Combine functions return a single key-value pair for each group of input pairs.

Putting all pieces together, a MapReduce program looks as follows.

$$
\begin{array}{rrl}
 & \texttt{InputFormat} \rightarrow & [(k_1, v_1)] \\
[(k_1, v_1)] \rightarrow & \texttt{Map(mapUDF)} \rightarrow & [(k_2, v_2)] \\
[(k_2, v_2)] \rightarrow & \texttt{Combine(combineUDF)} \rightarrow & [(k_2, v_2)] \\
[(k_2, v_2)] \rightarrow & \texttt{Reduce(reduceUDF)} \rightarrow & [(k_3, v_3)] \\
[(k_3, v_3)] \rightarrow & \texttt{OutputFormat} &
\end{array}
$$

The most important feature of the MapReduce programming model is its abstraction of data parallelism. Due to the functional origin of the parallelization primitives Map and Reduce, the user is not involved in any issues related to parallel execution. The user only provides a Map UDF and a Reduce UDF and the execution system guarantees to call these UDFs with the correct subsets of key-value pairs. Thereby, the mode of operation of the execution system is transparent to the user. Since all UDFs operate on independent subsets of the data, these calls can be carried out in parallel.

MapReduce Execution Model Similar to GFS, the MapReduce processing system follows a master-worker architecture. A single master and multiple workers are running on a compute cluster. Each worker offers a number of slots to execute tasks. The number of slots offered by a worker depends on the hardware resources of the machine it is running on, such as CPU cores and main memory. The master orchestrates the execution of a MapReduce program and schedules processing tasks to slots. For execution, the master generates M input splits based on the input format and M Map tasks to process the input in parallel. Each input split corresponds to one Map task. The number of Reduce tasks can be configured by the user.

All MapReduce jobs are executed with exactly the same processing strategy. Figure 2.3 depicts MapReduce's static execution model. After a MapReduce program has been started, the master begins to schedule Map tasks to workers with available task slots. Once a Map task is scheduled, the input format starts reading the data of the associated input split and generates key-value pairs. Each key-value pair is individually passed to and processed by a Map UDF. The key-value pairs that are emitted by the user-defined function are collected in memory. Periodically, the collected pairs are sorted and handed to the Combine function. The resulting pairs are partitioned by a hash function such as $hash(key)\ mod\ N$ into N buckets and written to disk. The Map tasks reports the location of the written partitions to the master which informs the Reduce task responsible for

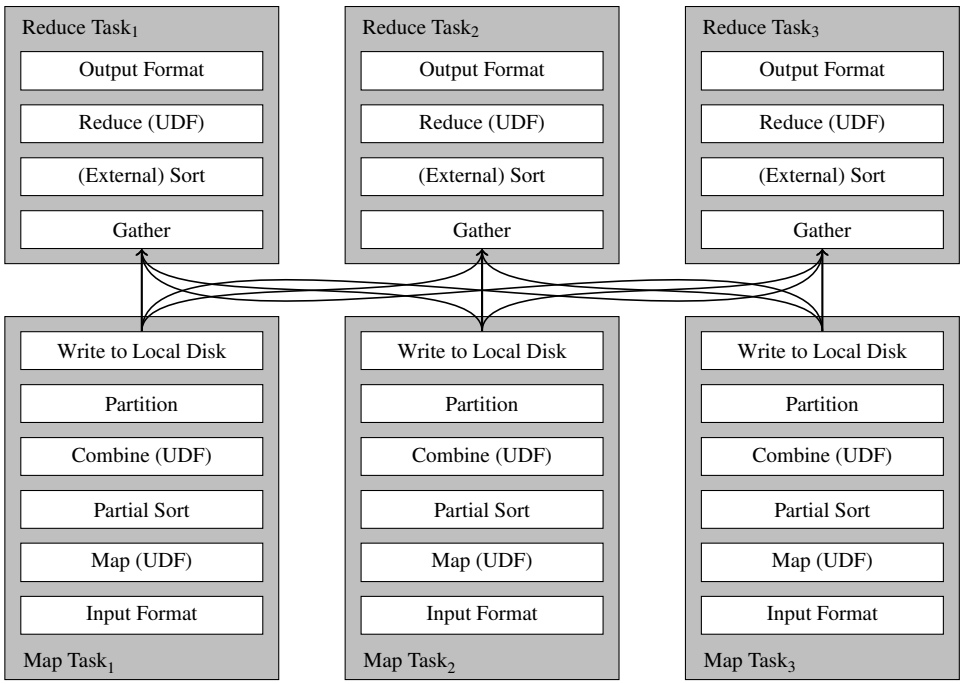


Figure 2.3: The MapReduce execution model, c. f. [127]

that partition about the location of its data. Once a Reduce task is scheduled, it pulls its partitions from all Map tasks over the network and locally sorts the key-value pairs by their key. In case the amount of data is too large for an in-memory sort, the sort is externally performed using the local hard disks. After sorting, the key-value pairs are grouped by key and the groups are handed one-by-one to the Reduce UDF. Finally, the output of the Reduce UDF calls is handed to the output format and the program execution finishes when the last Reduce task is completed.

It is not a coincidence that MapReduce's execution model integrates well with the distributed Google file system. When reading a file from GFS as input for a MapReduce program, the chunks of the file are translated into input splits which can be independently processed in parallel. When scheduling a Map task to an open task slot, the master aims to assign the task to a machine that is as close to the data as possible. Ideally, the Map task is executed on the same machine that also hosts the corresponding file chunk. If this can be achieved, the file can be locally read without incurring any network traffic. Due to the replication of chunks, local reads can be usually achieved for the majority of input splits. Note that MapReduce's data reading phase follows a shared-disk strategy since each worker can in principle read all data, while the remaining processing is carried out in a shared-nothing fashion.

The MapReduce execution model and processing system was designed to run on large clusters of commodity hardware. In such settings, failing task execution due to software or hardware

failures is rather common than an exception. MapReduce's system design is able to continue program execution in presence of such failures. The master checks for failures by pinging each worker. In case a worker does not respond within a certain time interval, the master marks its state as failed. All Map tasks and non-completed Reduce tasks which had been scheduled on this worker are rescheduled to other workers. Since Map tasks store their results locally on their hard disk, this data is lost in case of a machine failure. Hence, all Reduce tasks are notified about the failed Map tasks and their new execution location.

Higher-Level Programming Abstractions for MapReduce The combination of MapReduce and a distributed file system, such as GFS [92] or HDFS [109], yields generic data processing systems that are capable of processing large amounts of data. This is due to the file system storage abstraction which does not enforce any schema on the stored data and MapReduce's support of arbitrary data types and user-defined functions. However, this generality comes at a price. Providing well-performing implementations even for moderately complex tasks using the MapReduce programming model requires skill and effort.

In addition, data processing tasks often apply similar data processing operations and originate from similar domains, such as relational, extract-transform-load (ETL), and machine learning applications. Implementing such tasks using the plain MapReduce programming model induces significant code overlap and high implementation costs. This motivated the design and implementation of several higher-level programming abstractions on top of MapReduce. These abstractions provide domain specific operations to ease the definition of data processing tasks and compile their programs into MapReduce programs. Most of these abstractions were implemented using the openly available Apache Hadoop system [109]. In this section, we briefly introduce and discuss some of these systems.

Apache Hive [124, 198, 199] is a data warehousing system built on top of Hadoop. It was started as a project at Facebook and joined the Hadoop ecosystem as an Apache project later. Hive offers a SQL-like query interface and adapts storage techniques used by parallel relational database systems to provide storage formats which optimize the management of relational data on distributed file systems. Hive features a system catalog called Metastore that holds schema and partitioning information of the data imported and managed by Hive. From an interface point of view, Hive is similar to a parallel relational database system. However internally, Hive queries are translated into MapReduce jobs. Compared to database systems, Hive cannot leverage the benefits of relational processing engines, such as pipelining, but at the same time offers robust fault-tolerance, i. e., queries do not fail in case of hardware failures. Google's Tenzing [49] offers a SQL abstraction for Google MapReduce similar to Hive. Tenzing breaks at some point with the original MapReduce programming and execution model to avoid performance overhead.

Pig [90, 169, 175] and Jaql [28] are two other representatives of higher-level languages on top of Hadoop. While Pig is an Apache project, Jaql is part of IBM's big data platform InfoSphere

BigInsights [30]. Both languages operate on a semi-structured and nested data model and follow an imperative programming style, i. e., scripts are composed step by step. Each step performs a simple data transformation or operation, such as filtering, grouping, joining, or a user-defined function. The supported operators have been chosen with a focus on parallel execution. The imperative style is an obvious difference to SQL. Due to their step-wise programming style, extensive support for user-defined operations, and handling of semi-structured data, Pig and Jaql are well suited for complex data flow definitions, such as ETL processes.

Cascading [44] is a programming model to compose advanced data flows consisting of multiple data sources, operators, and data sinks. These data flows are compiled into sequences of MapReduce jobs. Cascading is based on the concept of pipes that are connected and chained into assemblies. Different types of pipes apply user-defined functions in different ways to data items. For example the Each pipe is semantically equivalent to a Map function and applies a user-defined function to each individual item. Further pipes support group-wise processing, joining, splitting, and merging. Cascading features a record-based data model. In contrast to higher-level languages, such as Hive, Pig, or Jaql, Cascading is a more low-level programming abstraction.

IBM's SystemML [93] features a domain-specific language for machine learning called Declarative Machine learning Language (DML). DML resembles the programming language R and provides linear algebra and mathematical programming primitives that manipulate matrices and scalar values. These kinds of operations are commonly used to implement a wide range of machine learning algorithms. In addition, DML offers loop primitives to implement iterative algorithms. DML does not offer the full set of R functionality but focuses on operations which can be effectively parallelized. SystemML optimizes algorithms implemented in DML and compiles them into workflows consisting of multiple Hadoop MapReduce jobs.

Using Apache Hadoop as a processing back-end for higher-level languages gives several benefits, such as excellent scaling and fault-tolerance behavior, but also has major disadvantages. Most data processing tasks implemented in a higher-level programming abstraction are compiled into workflows consisting of multiple Hadoop jobs. The execution of each job has high constant costs, which are due to scheduling overhead and the transfer of intermediate results via a distributed file system. These limitations of Hadoop motivated the design and development of novel data processing systems, such as Spark [190, 207, 208], Stratosphere [10, 23, 195], and Tez [197] which are based on a data flow abstraction[2]. Once these systems became more mature, a new trend evolved. Higher-level programming abstractions that were originally built on top of Hadoop are migrating to data flow systems. For example Hive and Pig have been ported to Tez and similar plans exist for Cascading. The adoption of a new processing back-end significantly improved the performance of Hive and Pig.

[2] Data flow systems are discussed in Section 2.1.4

Performance Improvements for MapReduce Prior to MapReduce, the predominant solution to analyze large amounts of data have been parallel RDBMSs. Compared to parallel RDBMSs, MapReduce's open-source implementation Hadoop allows to store large amounts of data at much lower cost and analyze it using a generic programming model in a scalable and fault-tolerant manner. However, it has been recognized that data processing using MapReduce can be inefficient. Especially for relational workloads, which were traditionally executed on relational DBMS, Hadoop's performance is often very low. Due to this reason, several approaches have been proposed to improve the performance of certain applications on Hadoop. Most of these approaches adapt techniques known from relational database systems and port them to Hadoop. In the following we list some works in this direction.

Several approaches have been proposed to improve the access to structured data. Google's Bigtable system [48] is a store for structured data that is built on top of GFS [92]. Data is indexed by three attributes, a row key, a column key and a time stamp. Bigtable offers point access as well as range access. Apache Hbase [115] an open source implementation of Google's Bigtable and is built on top of HDFS. Hadoop++ [70] and the Hadoop Indexing Library (HAIL) [71, 181] provide input formats for Hadoop, which are specialized for accessing structured data. Hadoop++ embeds read-optimized indexes into HDFS file chunks and features data co-location to improve join processing. HAIL leverages HDFS's replication by storing differently sorted replicas of file chunks. By doing so, each replica is read-optimized for a different access pattern. Further approaches to improve data access include the adaption of columnar storage formats [85, 153], data co-location [76], and scan sharing [167]. In addition to data storage, techniques have been proposed to improve the performance of iterative MapReduce workflows [41], Hadoop's scheduling behavior in heterogeneous environments [209], and support for pipelined execution of MapReduce programs [58].

Another class of MapReduce-based systems are hybrids of MapReduce and parallel database systems. HadoopDB [3] is based on Hadoop MapReduce and Hive. It addresses Hadoop's deficiency for sophisticated data storage techniques, such as indexes and columnar storage layouts. HadoopDB replaces HDFS by one non-parallel, stand-alone database system on each machine of the Hadoop installation. A query optimizer splits a SQL-like Hive query into queries that can be locally answered by the stand-alone DBMSs and a global data flow consisting of one or multiple MapReduce jobs. The goal of the optimization is to push as much processing as possible into the local database processors. Similar to parallel relational database systems, the amount of locally processable work depends on a proper data partitioning configuration. Polybase [69] follows another approach. It combines Microsoft's SQLServer Parallel Data Warehouse (PDW) and Apache Hadoop [109]. Polybase queries can jointly access relational database tables and files stored in HDFS. The optimizer automatically decides which parts of a query to execute using the MapReduce engine or SQLServer's relational processing engine.

2.1.4 General Parallel Data Flow Systems

MapReduce tightly couples a programming and an execution model. Both models are static and do not support to rearrange or add additional operators or processing steps. Although the data flow of a MapReduce program can be customized by user-defined functions [70], its overall structure is fixed. The MapReduce execution model can be understood as a hardwired parallel data flow. The fixed execution plan is a major performance issue for more advanced data processing tasks. Application logic that does not fit into a single MapReduce program needs to be split and implemented as multiple programs. In addition to being cumbersome to implement, the execution of multiple MapReduce programs results high scheduling overhead and low performance because data must be transferred jobs via a distributed data store.

In contrast to MapReduce's fixed execution model, there are a several systems which feature the execution of generic parallel data flows. These systems feature flexible and efficient execution engines for higher-level programming abstractions. In this section, we will present parallel data flow systems in detail, list some representatives, and discuss higher-level programming abstractions on top of these data flow systems.

Data Flow Program Abstractions Data flows are an intuitive abstraction to specify data processing jobs. In this abstraction, data is read from one or more data sources and "flows" from processing task to processing task until the desired result is computed and emitted to a data sink. Data flows are often realized as directed, acyclic graphs (DAGs). The vertices represent processing tasks, data sources, and sinks, and the edges represent connections between tasks over which data is transferred. Each processing task is a modular piece of sequential processing code which reads a data set or stream and produces a data set or stream.

Data flows can be executed in a distributed computing environment using task parallelism as well as data parallelism. For task parallelism, different processing tasks are simply assigned to different compute resources. Data parallel execution is a bit more intricate and requires that processing tasks are split up into multiple subtasks. Each subtask processes a subset of the original processing task's input data and can be assigned to a different compute resource. However, it depends on the semantics and implementation of a processing task whether it can be split into subtasks or not. In addition, a processing task might require a certain distribution of its input data to its subtasks in order to produce semantically valid results. For example a task that requires all data elements of a certain type to be processed by the same subtask needs a data-sensitive routing of data elements, i. e., data partitioning. Figure 2.4 shows a data flow (Figure 2.4(a)) and a parallelized version (Figure 2.4(b)). The original data flow consists of three data sources (Sources A, B, and C), four processing tasks (Tasks A, B, C, D), and a single sink. For the parallelized version, Source A and Task A have been split into three subtasks each. Sources B and C, Tasks B and D, and the sink have been split into two subtasks each and Task C is represented by four subtasks. Figure 2.4(b) shows as well that the connection patterns of subtasks differ

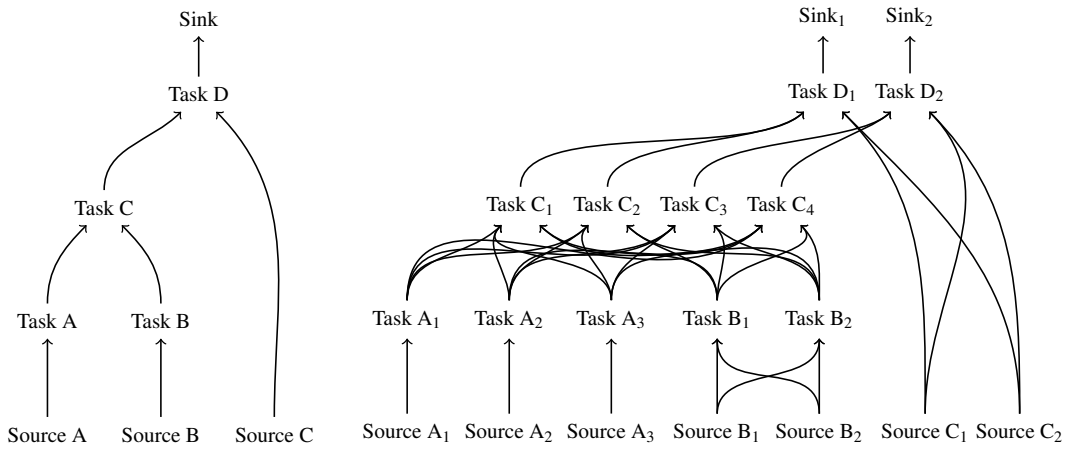


Figure 2.4: Parallelization of a data flow

depending on the semantics of the connected tasks. While each subtask of Source A feeds its data to exactly one subtask of Task A, each subtask of Source B is connected with every subtask of Task B for example to establish a partitioning of the input of Task B.

Execution of Parallel Data Flows Parallel data flow processors provide a programming abstraction and take care of task scheduling, data transfer between tasks, and fault-tolerance. However, systems differ in the techniques to provide these features. For example resource allocation and task scheduling can be achieved using built-in mechanisms, dedicated cluster schedulers, such as Hadoop Yarn [205] or Apache Mesos [123, 162], or Infrastructure-as-a-Service offerings. Data can be transferred using different types of communication channels, such as network connections, in-memory copying, and writing to and reading from files in local or distributed file systems. Also the mode of communication, i. e., blocking or pipelined data shipping, and the provided mechanisms to overcome failures differ among data flow systems. Representatives for data flow systems are Dryad from Microsoft [72, 133], UC Irvine's Hyracks [36], UC Berkeley's Spark, Apache Strom, and Nephele [203], which was developed at TU Berlin. Apache Tez [197] is a new data flow engine that aims to serve as a processing engine for higher-level languages and programming abstractions which are originally based on Hadoop MapReduce [109], such as Hive [124], Pig [175], and Cascading [44].

Higher-level Programming Abstraction for Parallel Data Flow Systems Parallel data flows offer an expressive and flexible abstraction to specify data processing tasks. Because this

abstraction is more flexible than SQL and MapReduce, many use cases can be implemented in a more straight-forward and efficient way without the restrictions of the MapReduce programming model or SQL's syntax. However, parallel data flows are a rather low-level abstraction such that this flexibility comes at the cost of usability. Programming abstractions for parallel data flows lack declarative task specification and require manual composition of parallel execution plans which essentially results in doing the job of a query optimizer. Implementing a well-performing parallel data flow requires expertise in parallel data processing and good programming skills and takes a significant amounts of time. The APIs of data flow processing systems are usually not designed for manual task specification. Instead higher-level programming abstractions have been designed which compile and translate their jobs into parallel data flows. In the following, we discuss some representatives of these higher-level programming models.

Mircosoft's DryadLINQ is a LINQ (Language INtegrated Query) compiler for Dryad [134, 206]. LINQ is an approach to tightly integrate data processing specifications into the higher-level programming languages of the .NET framework, such as C#, F#, or VisualBasic. Queries are stated in the same language in which the application is written using special LINQ libraries. In contrast, SQL is a dedicated and independent language and SQL queries are usually embedded as strings into application code. The integration of LINQ and the hosting programming language features characteristics of declarative and imperative languages. Embedded LINQ queries are compiled by so-called LINQ providers and executed against data processing systems, such as relational DBMSs, XML processors, or as in case of DryadLINQ against a parallel data flow system.

SCOPE is another high-level programming abstraction developed at Microsoft. It is a declarative query language, which heavily borrows from SQL [47, 212]. The data model is based on sets of rows consisting of typed attributes. The operator set and syntax resembles SQL. The language supports selection, projection, inner and outer joins, and aggregations. SCOPE queries can be written as nested queries or in an imperative style similar to Pig and Jaql, where multiple individual statements are linked. Nested subqueries are not supported. Similar to Pig, Hive, and Jaql, SCOPE is extensible and allows to define three types of parallelizable UDFs called Process (which is equivalent to Map), Reduce, and Combine, which correspond to the first-order functions of the MapReduce programming model. SCOPE features an advanced optimizer which compiles queries into data flows and executes them on Microsoft's Cosmos platform [47].

Spark is a parallel data processing system, which originates from UC Berkeley [208]. It became an Apache top-level project in 2014. Spark's execution model follows the model of parallel data flows. Its programming interface is based on the concept of resilient distributed data sets (RDDs) [207]. An RDD is a set of data items which are distributed over Spark worker nodes and ideally held in memory. By applying a transformation on an existing RDD, a new RDD is constructed. Spark supports a rich set of transformations including Map, Reduce, Filter, and Join and offers programming APIs for Scala, Java, and Python. In addition, physical data processing operators are supported, such as explicit in-memory caching and partitioning.

AsterixDB is a scalable information management system that operates on large clusters of commodity servers [15]. AsterixDB started as a joint project of UC Irvine, UC Riverside, and UC San Diego. The goal of the project is to store, index, process, analyze, and query semi-structured data [25]. Key features of the system are support for short and long running queries over structured and semi-structured evolving data sets. AsterixDB features a declarative query language called AQL (Asterix Query Language) which was designed to support a wide range of analyses and queries of semi-structured data sets. Query specified in AQL are optimized and compiled into Hyracks data flows for execution.

The PACT programming model, which was developed as part of this dissertation, belongs as well into the category of higher-level programming abstractions for data flows and will be presented in Chapter 3.

2.1.5 Comparing Parallel Data Processing Systems

In the previous three sections, we presented and discussed three classes of parallel data processing systems: 1) parallel relational database management systems, 2) MapReduce-based systems, and 3) general parallel data flow systems. In this section, we discuss the benefits and drawbacks of these approaches and compare the different classes.

Parallel RDBMSs have been used to process data sets which exceed the capabilities of a single machine for many years. Database systems provide data independence, enforce a relational data model, and offer a declarative query interface. Sophisticated query optimizers transform complex SQL queries into efficient execution plans, which are evaluated by extremely specialized processing engines. On the other hand, parallel RDBMSs have a number of drawbacks which helped MapReduce and in particular Hadoop to become popular. A relevant aspect of parallel database systems is their high cost of operation. Since these systems are not easy to manage, a significant fraction of operation cost is required to employ qualified personnel. Licensing costs and the cost of high-end hardware add to this. In addition, parallel database systems often assume a pool of homogeneous machines, which renders scaling out a challenging task. Putting economic arguments aside, relational database systems are not a good choice for data that does not nicely fit the relational data model. And even if data is relational, importing it into a database is often a too time consuming process. Moreover, new use cases, such as graph analytics or machine learning applications, cannot be easily expressed using SQL and would require cumbersome workarounds using external user-defined functions. In order to add support for emerging data analysis applications, database systems started to incorporate characteristics of MapReduce-based systems such as support for MapReduce-style user-defined functions [87] or massively parallel setups on commodity hardware clusters [201]. Further approaches integrated Hadoop and database systems such as for example HadoopDB [3] or Polybase [69].

MapReduce and in particular Hadoop addressed several of the short-comings of relational database systems with respect to new emerging use cases. One of the most important benefits of Hadoop is its lower cost of operations. The software itself is publicly available under Apache version 2 license and runs on heterogeneous clusters of commodity hardware of any size. New machines can be added or removed at any time allowing the cluster to dynamically scale. Platform-as-a-service offerings such as Amazon Elastic MapReduce [77] allow users to operate large Hadoop clusters on demand and mitigate the need for system administrators. High-level languages such as Hive and Pig are easy to learn and reduce expenses for user training. Another reason for MapReduce's popularity is its flexibility. The Hadoop Distributed File System stores data in any format, can be easily scaled, is fault-tolerant, and provides good read performance through parallel I/O. Furthermore, the generic MapReduce programming model eases the processing of any kind of data by supporting custom data types and user-defined functions. Moreover, Hadoop offers better fault-tolerance than most database systems, which are rather designed to avoid failures than to cope with them[3]. However, these benefits come at the cost of inefficient processing for certain kinds of workloads. Especially workloads that require several MapReduce jobs, such as relational queries or iterative tasks perform poorly on MapReduce systems. This is due to two reasons. First, the high costs of reading data from and writing data to a distributed file systems at the beginning and end of each jobs, and second, the high scheduling overhead caused by Hadoop's communication model. These deficiencies were also recognized by the database research community [173, 194]. Nonetheless, the open source community established a large ecosystem around Hadoop including several higher-level languages (Pig [175], Hive [124], Cascading [44]), data stores (HBase [115], Accumulo [4], Parquet [172]), management tools (Ambari [13], Bigtop [29]). Hadoop has achieved wide industry adoption with products of major IT companies, such as IBM, Microsoft, Yahoo!, supporting it. In addition several companies that package Hadoop distributions, offer support for the Hadoop ecosystem, and build commercial tools or custom solutions around it established.

Parallel data flow systems aim to close the gap between parallel RDBMSs and MapReduce-based systems. In fact, a parallel RDBMS is a parallel data flow system with a domain-specific programming front-end language (SQL) and the execution plans compiled by its optimizer are parallel data flows. On the other hand, the MapReduce execution model is a fixed parallel data flow consisting of a single data source, followed by a Map and a Reduce operator and finally a data sink. Parallel data flow systems aim to provide the scalability, fault-tolerance, and flexibility of MapReduce and performance-wise more similar to RDBMS than to MapReduce-based systems. Compared to the MapReduce programming model, programming models on top of data flow systems ease the definition of complex data processing tasks by providing an extensive set of operators which can be freely connected.

[3] Parallel RDBMSs are usually operated on smaller clusters of homogeneous high-performance machines with redundant hardware configuration. This reduces the probability and impact of hardware failures. Data loss due to disk failures is avoided by redundant storage systems, such as RAID setups.

2.2 Optimization of Parallel Data Flows

Parallel data flows are a flexible abstraction to define and execute data processing tasks. However, this abstraction is not well-suited for a broader audience since manual specification of parallel data flows requires expert knowledge in distributed data processing and good programming skills. To overcome this limitation, a wide variety of high-level programming abstractions which are compiled to parallel data flows have evolved. Among these abstractions are query languages such as SQL, Pig [169, 175], Hive [124, 198], and Scope [212], but also parallel programming models including MapReduce [64], Spark [190, 208], Cascading [44], and DryadLINQ [206]. The compilation of these abstractions into parallel data flows and their execution offers much optimization potential.

In this section, we briefly describe different approaches to compile higher-level program abstractions into parallel data flows and optimize their execution. We start presenting traditional cost-based optimization of relational queries and discuss problems with the state-of-the-art. We continue to discuss optimization techniques for plain MapReduce jobs and finally present the optimization of higher-level languages and parallel programming models. This section is based on our survey on the optimization of massively parallel data flows [127].

2.2.1 Optimization of Relational SQL Queries

Query optimization is one of the most extensively researched topics in the field of database systems. The popularity of SQL and the relational data model necessitated and powered the advances of query optimization. Declarative SQL queries define the desired result instead of imperatively specifying the execution steps. Therefore, it is up to the database system to decide how the requested result should be computed. It is the task of the query optimizer to find the most efficient execution plan for a given query. In this section we give a short introduction into cost-based query optimization and discuss open problems with this approach. Finally, we review special optimization techniques for parallel RDBMS.

Cost-based Query Optimization Most modern database systems feature a cost-based query optimizer that follows a three-staged approach [108]. First the query is parsed, syntactically checked, and translated into an internal representation. Second, *logical optimization* is applied. This step rewrites the internal query representation by reordering or replacing logical operators. Common rewrite rules in this step are selection and projection push-down, in-lining of logical views, and un-nesting of subqueries. Since these rules are either mandatory or expected to generally improve query performance, they are greedily applied whenever possible. The third step is called *physical optimization* and generates a physical execution plan. This process includes

the choice of physical operators and access paths and determines the order of joins[4]. There are two common approaches to enumerate plans in physical optimization, bottom-up and top-down. Bottom-up approaches enumerate candidate execution plans by starting at the base relations and finishing at the root of the execution plan. Commonly, dynamic programming techniques are used to construct an optimal plan from optimal subplans [163, 186]. In contrast, top-down enumeration techniques start at the root of an execution plan and move towards the base relations of a query. Branch and bound algorithms are often used to prune large fractions of the search space [82, 97]. All techniques rely on the ability to compare two equivalent (sub-)plans by estimating their execution costs. Since these costs are used to select plans and prune areas of the search space, high accuracy of cost estimates is crucial.

Optimizers feature a so-called cost model to estimate the execution cost of a plan. A cost model is a collection of functions that compute a cost metric such as resource consumption or execution time for a specific operator and a given execution condition. The execution condition of an operator includes, among other things, the amount of data to processes and the available memory budget. It is specified by the input parameters of the operator's cost function. Depending on the operator type these input parameters include statistics of the input data (e. g., cardinalities, data size, page counts, and value distributions), query properties (e. g., local and join predicate selectivities), and the amount of available resources (e. g., amount of memory, disk I/O and network bandwidth, and CPU utilization). Given a plan, the optimizer assesses the execution conditions for each operator and calls the corresponding cost function. The total cost of the plan is usually computed as the sum of the individual operator costs. During the plan enumeration process, the optimizer compares equivalent plans based on their estimated cost. Eventually, the plan with the least estimated costs remains and is executed.

Challenges of Cost-Based Optimization A major problem of cost-based query optimization is that input parameters of cost functions are usually not known at optimization time and must be estimated as well. That means that the optimizer needs to predict the conditions under which a query will be executed including available resources and properties of the data. In the following, we will discuss the challenges of estimating the input parameter of cost models and the consequences for cost-based optimization.

In disk-based database systems, the performance of a query significantly depends on the amount of data that needs to be read from and written to hard disks. Database systems try to reduce the number of physical I/O operations by caching data in memory. Since memory is a shared resource, the usage of memory must be coordinated across concurrently running queries. Consequently, the amount of memory that is available for a query at execution time varies with the query load of a system. The same is true for the amount of CPU and I/O resources and renders

[4] Although choosing the order of joins conceptually is a logical rewrite, it is carried out as part of the physical optimization because the optimal order depends on the choice of the physical join operators.

reliable predictions for the amount of query execution resources difficult. This issue is especially relevant if a query is not immediately executed, such as pre-compiled queries with parameters.

The amount of data to process is another crucial input parameter for optimizer cost models. Since the schema of the data is usually well-known in relational database systems, the problem boils down to estimate the number of records. This is called cardinality estimation. The number of records that an operator has to process depends on the operator tree blow its input. This tree consists of base relations and processing operators. The cardinality estimate for the tree's result is computed using the cardinalities of the base relations and the selectivity of all involved operators. While the cardinality of the base relations is typically known, the selectivity of the operators needs to be estimated. The selectivity of an operator depends on the operator itself and on the data that is processed. Since the semantics of relational operators are known, it is important to know certain properties of the input data, such as attribute value distributions. Database systems feature statistics stores to collect and provide such information. However, it is not feasible to collect and maintain all interesting statistics. This is especially true for statistics on multivariate value distributions. Optimizers fall back to default assumptions, such as the uniformity assumption, the independence assumption, and the inclusion assumption if certain statistics are not available [186]. However, these assumptions can be significantly off and cause estimation errors of several orders of magnitude [131, 164, 177].

In summary, cost estimates for query execution plans depend significantly on the accuracy of cost model parameters, which are hard to assess and can be off by orders of magnitude. This is a major problem for cost-based query optimization because not every plan is affected by this problem at the same magnitude. While some plans are very sensitive for changing sizes of intermediate results and suffer from significantly increasing execution time, other plans may behave much more robust with no or low performance regression. In fact, cardinalities of conjunctive predicates are frequently underestimated, which favors plans that excel at executing small amounts of data but perform horribly for large inputs.

In prior work, basically three classes of approaches have been proposed to tackle this problem. The first class aims to handle the problem within the optimizer, by identifying query execution plans that are "robust" with respect to uncertain information, i. e., imprecise cost model parameters. Early work in this direction is least expected cost optimization [55, 56]. The authors propose to use probability distributions instead of point estimates for cost estimation and compare execution plans based on their expected cost. Babcock et al. [17] propose a tuning knob for users to trade execution performance for predictable execution times. Chaudhuri et al. [50] focus on reducing the variance of execution times of parametrized queries. The second class proposes adaptive query evaluation methods. Instead of relying on the optimizer to identify a good execution plan, the runtime system is able to dynamically adapt the execution plans of queries to improve the evaluation performance. However, the set of possible adoptions is often limited by the initially chosen execution plan. Deshpande et al. [66] give an extensive overview of work in this direction. The last class uses hybrid techniques of both prior classes, i. e., it combines

uncertainty-aware optimization and runtime adaption. The optimizer injects check operators at critical positions into execution plan. Check operators track the size of intermediate data and compare it to a validity range specified by the optimizer. If the validity range is violated, the check operator triggers a reoptimization of the plan [159], or switches to plan alternative that was computed before [19].

In Chapter 5 we discuss the problem of "risky" query execution plans in detail. We present results of an experimental study to assess the sensitivity of query execution plans with respect to changing execution conditions, propose a metric to measure this sensitivity, and describe an approach to predict the sensitivity of execution plans for changing input sizes, and a method to prevent risky plan choices.

Optimization of SQL Queries for Parallel Execution Query optimizers for shared-nothing parallel database systems must take several additional aspects into account compared to optimizers of stand-alone DBMS. These aspects increase the search space of possible execution plans [68]. Parallel execution plans consist of subplans that are concurrently executed on multiple machines and connected by data reorganization steps such as data (re-)partitioning or replication [95, 96]. Multiple database instances process a query in parallel and ship partial results over the network in order to compute the final result. In contrast to a centralized system where the costs for disk I/O are predominant, network I/O costs usually account for the biggest portion of the absolute costs in parallel database systems. Therefore, cost models must take shipping costs and concurrent execution into account [143]. In order to reduce the amount of data shipped over the network, several techniques have been proposed. Data placement strategies such as partitioning, replication, and co-location enable the local computation of expensive operations such as joins and aggregations [143]. Due to these techniques large fractions of a query can be processed without any network communication. Reasoning about existing partitionings can significantly reduce the amount of shipped data and improve the query execution time. Further techniques to reduce network traffic are semi-join reduction and partial aggregation. Semi-join reduction improves distributed joins and reduces relations prior to shipping by filtering out tuples which will not match in the join [27, 143]. Partial aggregation is similar to MapReduce's Combine function and reduces data sets which are shipped by locally applying partial aggregation functions.

There are two common approaches to generate parallel query execution plans. The first approach is based on two phases, where the first phase generates a sequential plan which is subsequently translated into a parallel plan. The sequential plan can be obtained from an optimizer which is not aware of the subsequent parallelization [125] or an enumerator that targets plans which can be translated into well-performing parallel plans [113, 114]. In contrast to the two-step approach, there has also been work to directly compile parallel execution plans by incorporating parallelism into plan enumeration strategies and cost models [137, 191]. Since this approach has to deal with a significantly larger search space, also randomized search strategies have been proposed [146]. Mitschang et al. [166] refer to further relevant work in this context.

2.2.2 Optimization of Plain MapReduce Jobs

In contrast to SQL queries, MapReduce programs offer much less optimization potential. This is due to several reasons. First, a MapReduce job is specified as user-defined functions which impedes reasoning about its semantics. Lack of schema information and data statistics prohibits most cost-based optimization approaches. Using a distributed file system to store data complicates the application of sophisticated storage and data placement approaches, such as indexes and data co-location. Finally, MapReduce's hard-wired execution strategy significantly reduces the degrees of freedom for physical optimization. However, some MapReduce systems, such as Hadoop, provide more interfaces than Map and Reduce and several configuration parameters. In the past, techniques and approaches have been proposed to improve the performance of MapReduce jobs. Most work in this direction is based on Hadoop MapReduce. Not all techniques can be transparently applied, but require user interaction. Among the proposed approaches are techniques to improve data access, such as indexes [70, 71], columnar layouts [85, 153] and data co-location [76] and iterative programs [41]. In this section we present and discuss some optimization techniques for MapReduce programs which can be transparently applied.

Analyzing and Modifying MapReduce UDFs A large problem when optimizing MapReduce programs is the hard-wired execution plan. In essence, every MapReduce program is executed in exactly the same way. All job specific logic is contained in user-defined functions. It might be possible to optimize a MapReduce program by improving the implementation of its user-defined functions if their semantics are preserved. However, user-defined functions are usually implemented in an imperative programming language, such as Java, which causes difficulties for an optimizer to infer the semantics of a job by looking at the code.

Fortunately, there are a few data processing operations which are commonly expressed in easy-to-detect code patterns, such as selection and projection. These operations can reduce the amount of data to process and have much potential for performance improvements. In relational database systems, selection and projection operations are pushed as close to the data source as possible for this reason. Further, the use of indexes can significantly reduce the number of physical I/O operations and is among the most important optimization in database systems. Manimal [42, 136] employs static code analysis to identify selection and projection code patterns in Map and Reduce functions. Based on its findings, the user code is transparently modified to read data from an index instead of performing a full scan. The analyzer follows a conservative policy and only reports optimization opportunities that can be clearly identified. Manimal's optimizer applies simple rule-based heuristics [42]. It inspects all optimization opportunities identified by the analyzer and tries to match these with available indexes retrieved from a catalog. Based on this information, the optimizer chooses the index that promises the best performance gains. Conflicting optimization choices are solved with prioritization. Most of the proposed optimizations

are based on indexes. Along with optimization opportunities that require an index, the analyzer emits an index creation program to create the necessary index from the original job input data.

Manimal's optimizations are well-known from relational database systems. Its contribution comes from the fact that these optimizations are applied to arbitrary user code instead of semantically rich declarative queries. Experiments with third-party user code [173] show that Manimal's analyzer identifies most optimization opportunities in relational-style jobs [136]. Currently, the system is restricted to individual MapReduce jobs. Workflows consisting of multiple jobs cannot be optimized as a whole. Also, the decision to create an index is not made by the system. Instead, the user has to choose whether and which indexes to create. An index advisory component as known from relational database systems [5] is not part of Manimal.

Job Profiling and Configuration Tuning The execution of a MapReduce program follows a fixed strategy which cannot be adapted. However, the performance of a Hadoop job also is influenced by configuration parameters, such as the degree of task parallelism, the size of sort buffers, and fill ratio to spill a sort buffer. In total, Hadoop provides more than 190 parameters of which about 25 can have significant impact on the performance of a job [18]. In database systems, some of these parameters are automatically chosen by the optimizer, such as the amount of memory for sort operations or the degree of parallelism for tasks. Hadoop does not offer any support in choosing the "right" parameter values. Instead, there is much literature about Hadoop configuration which is based on best practices, experience, and empirical evidence. However, manual tuning of these parameters is difficult since the optimal configuration depends on the job, the input data, and the available compute resources. Moreover, some parameters interact with others.

Starfish [121, 122] is an approach to optimize the execution of plain Hadoop MapReduce programs by properly configuring the Hadoop environment and the program. Prior to its execution, a Hadoop job is intercepted and analyzed. Based on this analysis, Starfish generates optimized parameter settings for Hadoop MapReduce jobs. Starfish consists of three major components: a job profiler, a What-If engine, and a cost-based optimizer to improve the execution of arbitrary MapReduce jobs. Prior to optimization, the profiler monitors the job's execution to derive its performance characteristics. For this purpose, the Hadoop MapReduce execution pipeline is split up into 13 phases [121]. During profiling the job's behavior in each phase is monitored and job profile with more than 60 gathered metrics is generated. The job profile is handed to the What-If engine which features a cost model for Hadoop's execution model [120]. The engine is able to simulate the execution of the job for varying environmental conditions, such as different input sizes, compute resources, and parameter settings. The cost-based optimizer has to explore a high-dimensional, nonlinear, non-convex, and multimodal search space of parameter settings. For each candidate configuration, the optimizer obtains a cost estimate by calling the What-If engine. It also uses two techniques to split and explore the space of all possible configurations [121]. First, the highly dimensional search space is split into multiple subspaces

with fewer dimensions. Many parameters influence only a certain part of the whole MapReduce execution pipeline, such as the Map or the Reduce phase. Parameters that interact or effect the same parts of the pipeline are clustered. The optimizer processes all clusters independently from each other. Second, the optimizer employs a technique called Recursive Random Search (RRS) to find good parameter settings within a cluster. RRS starts picking arbitrary points in the search space. For each point, the What-If engine is called and computes a cost estimate. In consecutive iterations, the optimizer inspects promising areas in more detail by taking more samples from them. Finally, the optimizer returns the configuration with the shortest estimated execution time found so far.

Starfish is an optimization framework that is generally applicable for arbitrary MapReduce jobs. Even jobs which are generated by a high-level language compiler, such as Pig or Hive, are supported. An evaluation shows a high accuracy of the What-If engine's cost estimates and a good quality of the optimizer's configuration choices [121].

Optimizing MapReduce Job Workflows One of the drawbacks of MapReduce is that application logic needs to be fit into the tight corset of the MapReduce programming model. It is common that data processing tasks cannot be implemented as a single MapReduce job. Instead the application logic needs to be spread over multiple jobs. Depending on the interdependencies of the jobs, these jobs can be executed in parallel or must run sequentially. Scheduling multiple MapReduce jobs is costly due to the high start-up overhead for task scheduling and coordination. Moreover, the input of a MapReduce job forwarded to a subsequent job by writing it to and reading it from a distributed file system which is also expensive. Consequently, a major performance optimization when executing complex data processing tasks is to reduce the number of required jobs as much as possible. However, this requires experience and leads to non-modular and clumsy program code.

Stubby [151] is a transformation- and cost-based optimizer for workflows consisting of multiple MapReduce programs. The optimizer relies on manual annotations and optimizes workflows by merging MapReduce programs together in order to reduce the overhead of executing multiple MapReduce programs. In particular, Stubby features four optimization transformations, 1) intra-job vertical packing which merges a Reduce function into its preceding Map function, 2) inter-job vertical packing which merges two subsequent MapReduce jobs, 3) horizontal packing which merges two jobs that consume the same input data, and 4) partition function transformations which manipulate the partition function of a job to either enable other transformations or to remove data skew. Stubby requires annotations to reason about possible optimizations. These annotations include physical data properties such as partitioning, sorting, and compression, schema information for input and output key-value pairs of programs, and performance annotations. The optimizer uses two techniques to generate candidate workflows. First, it partitions a workflow into optimization units of jobs which can affect each other. Second, it applies transformation rules in a predefined order. The optimizer leverages Starfish's what-if engine

to obtain cost estimates. Since plans are merged, Stubby needs to adjust Starfish's job profiles by merging them. Stubby's approach is transparent to the origin of the workflow. It is able to optimize workflows which are manually composed, compiled from higher-level languages, such as Pig or Hive, or data flow programming models, such as Cascading. In case of compiled data flows, the compiler usually has all required information to also attach the required annotations on the generated MapReduce jobs. An evaluation shows that workflows which were optimized by Stubby consistently outperform non-optimized Pig programs by a factor 2 to 4.5.

2.2.3 Optimization of Higher-Level Programming Abstractions

Implementing an advanced data processing program as a MapReduce program or a parallel data flow requires extensive knowledge in parallel data processing, good programming skills, and finally takes much time. Several higher-level languages and programming abstractions have been designed to ease the definition of parallel data processing tasks. Tasks implemented in such a higher-level abstraction are compiled into MapReduce programs or parallel data flows and executed as such. This compilation step is conceptually similar to query optimization in relational DBMSs. Similar as for declarative queries, the compilation of data processing tasks implemented in a higher-level language or programming abstraction offers degrees of freedom which can be leveraged for program optimization.

When discussing the optimization of data processing tasks, we need to distinguish two distinct questions. First, what are the degrees of freedom for optimization, and second, which are the most beneficial optimizations. The first question addresses the search space of an optimizer, i. e., the set of all execution plans the optimizer can choose from. The second question addresses the problem of finding the best plan within this search space. In the context of relational query optimization, the first aspect is addressed by a well-known set of logical rewrite rules and different physical implementations for logical operators which span the optimizers search space. The optimizers plan enumeration technique and its cost estimation framework which includes its cost model and cardinality estimator, takes care of finding the best plan within the search space.

In this section, we discuss optimization techniques for higher-level programming abstractions. We start by comparing the optimization of higher-level programming abstractions for parallel data processing to traditional relational optimization. Subsequently we discuss the optimization techniques of higher-level languages which are compiled to MapReduce programs and some advanced programming abstractions which are translated into parallel data flows.

Differences between Optimization of Relational Queries and Higher-Level Programs

The optimization of programming abstractions for parallel data processing differs from relational optimization in several aspects. Most languages and programming models feature relational operators, such as selection, projection, and join. Consequently, several rewrite rules known from

relational query optimization can be applied, such as reordering two join operators. However, most programming abstractions also offer extensive support for user-defined functions. UDFs can pose a challenge for query optimizers, because their exact semantics are not known to the system. To what extent this is a problem depends on the interface of the UDF. For example, a UDF operator may still be pushed down if the optimizer knows that it implements a user-defined filter function. On the other hand, such a plan rewrite would not be possible if the operator only knows about a generic Map function that may emit a random number of arbitrarily modified records for each input record. Hence, limited information about user-defined functions reduce the size of an optimizer's search space.

In addition also the question, whether an optimization should be applied or not is more difficult to answer in programs with UDFs. Cost-based relational optimizers rely on the ability to estimate the costs of execution plans to guide their plan choice. This approach requires accurate estimates of intermediate result cardinalities. Relational database systems require to load data before it can be queried. During this (often costly) loading process, most DBMSs gathers basic statistics, such as table cardinalities. More detailed statistics can be collected to further improve the accuracy of cardinality estimates. In contrast to relational database systems, most higher-level languages and programming abstractions do not come with a dedicated data storage layer or metadata catalog. Instead, they aim to support the analysis of in-situ data which can be read from different data stores, such as distributed file systems, key-value stores, or database systems. Consequently, these systems do usually not require an expensive data loading step. This comes at the cost of lacking valuable cardinality information, values and distribution statistics, and possibly also schema information at optimization time. Cost estimation is even more exacerbated by use of user-defined functions which may produce an arbitrary number of output records or have unknown costs to process an input record.

Another opportunity for optimization is the choice of physical execution strategies. For example a join operator can be executed using a sort-merge join strategy or a hash join strategy. Note that these execution strategy choices also exists for operators that include user-defined functions.

Optimizing Higher-Level Programming Languages for MapReduce Despite the reduced optimization potential and the limited information to carry out deliberate optimization decisions, there are a few techniques that can be applied to optimize data processing programs. Most higher-level languages feature heuristics-based optimizers which apply transformations that are considered to generally improve the performance [28, 90, 198]. Depending on the concrete system, those rules include selection and projection push-down, variable and function in-lining, nesting, un-nesting, and field access rewrites. For example, the Jaql optimizer features more than 100 rules which are greedily applied. In order to preserve optimization potential in presence of user-defined functions, Jaql also supports explicit annotations, which reveal certain characteristics of user code that can be exploited by the optimizer [28]. The choice of physical operators, such as sort-based or hash-based aggregation and Map-side or Reduce-side join

implementations [33], is often left to the user. Optimizer hints are commonly supported and allow to choose and configure execution strategies [90, 198]. The order of joins is usually not optimized due to lack of data size information. Instead, relations are joined in the order that was specified by the user. Due to its metadata store, Hive is able to limit table scans to relevant partitions.

Another aspect to consider when discussing the optimization of higher-level languages is the execution plan abstraction to which a query or program is compiled. Hive, Pig, Jaql, and Cascading leverage Hadoop MapReduce as a processing back-end. However, as previously noted, MapReduce only offers a rather static corset into which programs need to be fitted. On the one hand, this reduces the number of possible execution plans compared to a compilation onto a general data flow system, but also requires special optimization techniques. A crucial aspect when generating workflows of MapReduce programs is to minimize the number required programs as much as possible. Usually, this heuristic produces good results since each MapReduce job comes with high overhead costs resulting from scheduling overhead and the fixed execution pipeline of reading the input, sorting, shuffling, and writing the result. Higher-level language compilers use techniques to chain several record-wise operations into a single Map function [168] or append them at the end of a Reduce function. Hive transforms multiple binary joins on the same attribute into a multiway join [199]. Both, Hive and Pig use a technique to multiplex independent operations on the same data into a single MapReduce program [90, 198].

A feature called "physical transparency" differentiates Jaql from Pig and Hive [28]. Low-level physical operators and high-level declarative operators are conceptually the same in Jaql. Both types of operators can be mixed when writing programs. That allows users to force the execution of certain strategies and eases the implementation of own operators using lower-level constructs. In addition, the result of optimizing a Jaql script is again a valid Jaql script, which significantly eases debugging.

Higher-level programming abstractions on top of Hadoop MapReduce have become popular tools to process large amounts of un- and semi-structured data. However, the high overhead of executing multiple MapReduce jobs for a single program has motivated a few trends. First, higher-level languages, such as Pig, Hive, and Cascading, are ported to data flow processing engines, such as Apache Tez [197] and Apache Spark [190]. Second, optimized storage formats, such as Parquet [172] and ORC (as part of Apache Hive [124]), significantly improve data access. These formats use indexing, compression, and columnar storage techniques to improve data access. Moreover, they also provide a certain statistics. Hive extends its metadata storage to also include more detailed data statistics. This evolution clearly improves the optimization capabilities of these systems. In fact, there are efforts to add a cost-based optimizer to Hive. Although this step is a significantly improvement of Hive's optimization capabilities, it should be noted that the optimization techniques used will be similar to traditional relational query optimization [67, 68].

MRQL [81] is a SQL-like query language and an advanced optimization framework that translates nested queries into workflows of MapReduce programs. By providing an expressive nested query language, MRQL aims to eliminate the need for opaque user-defined functions as commonly required by applications which are implemented for Hive or PigLatin. Due to this algebraic approach, the full query semantics are known to the optimizer. MRQL features a cost-based optimizer and novel translation techniques. MRQL was open-sourced and has entered the Apache Incubator [165]. Since then, MRQL has evolved and supports additional backends for query execution, such as Apache Hama [111], Apache Spark [190], and Apache Flink [84].

Optimizing DryadLINQ Programs The DryadLINQ [73, 206] provider extracts data parallel LINQ code from applications and compiles it into Dryad data flow programs. DryadLINQ's optimizer shares several features with traditional optimizers of parallel relational DBMSs [206]. It is based on greedy heuristics and applies optimizations such as pipelining of operations, minimizing repartitioning steps, eager aggregations, and reduction of I/O by using TCP-pipe and memory-FIFO channels. The optimizer derives much semantics from the program code by static typing, static code analysis, and reflection. Using these techniques, it reasons about prevalent data properties, such as partitioning and sorting. Partitioned data stores are supported as well. DryadLINQ features a rich set of user annotations to hint potential optimizations and expected memory consumption of user code. A notable feature is dynamic optimization. DryadLINQ adapts execution plans at runtime in order to perform network-aware aggregations, tree broadcasting, and determines partitioning properties, such as number of partitions and key-ranges for range-partitioning to handle skewed data [141, 206].

Optimizing Scope Programs Microsoft's Scope is a higher-level language on top of the parallel Cosmos execution engine [47, 212]. It features a sophisticated cost-based optimizer framework that uses several novel techniques to improve the execution of parallel data flows. The optimizer is based on the Cascades framework [97]. Scope features several different execution strategies known from parallel database systems, such as alternative join algorithms and sort- or hash-based grouping. The optimizer includes transformations for selection and projection push-down, eager aggregation, and chooses appropriate partitioning schemes. A special focus was put on optimizing partitioning, sorting, and grouping [213]. These data properties are crucial for data parallel processing, required for many operations, and expensive to establish. Minimizing the number of partitioning, sorting, and grouping steps is a major optimization goal for parallel environments. When reasoning about existing data properties, the Scope optimizer takes functional dependencies and constraints on the data into account. Moreover, Scope features static code analysis (SCA) to analyze user-defined functions and infer whether existing data properties are preserved after a UDF has been applied [211]. While this approach is non-invasive, i. e., it does not modify the user code, another technique applied automatically adapts

the user code to inject projections of fields which are not accessed in UDFs and moves filter conditions across UDFs to reduce the amount of processed data [106]. Scope addresses the problem of missing data statistics by online statistics collection. During optimization, the optimizer automatically injects statistics collection operators into a program to obtain missing statistics, such as cardinalities, average record sizes, value distributions, and UDF execution costs. Statistics are gathered and stored in a statistics store. A program signature allows the optimizer to retrieve relevant statistics later from the store. The collected statistics are leveraged in an offline and an online way by Scope. First, they are used to improve the optimization of identical or similar programs which are later optimized [38]. Second, statistical information is also used to optimize a currently running program from which it was collected. Whenever, new statistics are retrieved by the optimizer, it reoptimizes the data flow of the currently running program and decides whether it needs to be adapted or not [40]. Given all these features, Scope has the most advanced optimizer for massively parallel data flows that we are aware of.

2.3 Summary

In this section, we discussed three classes of systems for parallel analytical data processing and their capabilities to optimize queries and programs. While relational DBMSs offer an easy-to-use interface and high processing performance, they are not flexible enough for many of today's data analysis tasks. On the other hand, MapReduce systems lack performance and ease-of-use. Due to these limitations, novel distributed data processing systems have evolved to close the gap between both system families. Some of these systems are based on parallel data flows which are a flexible abstraction for efficient data processing. However, manual specification of parallel data flows is a cumbersome task and requires a rare skill set.

Consequently, there is a need for higher-level languages and programming models on top of parallel data flow systems which are easy to use and expressive enough to meet the requirements of today's data analysis applications. Program optimization is a fundamental building block to achieve these goals because it hides much of the complexity of a parallel and distributed data processing system from users and significantly improve processing efficiency and performance. However, program optimization suffers from lack of program semantics as induced by today's general purpose programming abstractions. In the following chapters of this thesis, we propose an expressive programming abstraction with declarative characteristics to define parallel data processing tasks and address challenges that arise in the context of data flow optimization.

3 Abstractions for Parallel Data Flows

Contents

In order to support today's data analysis applications, data processing systems must cope with rapidly growing data set sizes and support a wide range of analysis tasks on possibly unstructured ad-hoc data. A common solution to analyze large data sets is parallel processing. Advanced analysis methods can be applied to unstructured data due to support of user-defined functions and user-defined data types. MapReduce [64] is the most prominent and popular approach that meets these requirements. It provides a programming model and a scalable execution model to analyze large amounts of data in a massively parallel and fault-tolerant way.

However as noted in Chapter 2, MapReduce has also several deficiencies. While in principle being expressive, MapReduce is not well suited for complex data analysis tasks. Applications that do not fit the programming model need to be implemented as a workflow of multiple MapReduce programs. Joint analysis of two or more data sets requires unintuitive workarounds and common operations such as relational joins need to be manually implemented. In general, implementing data analysis applications using the MapReduce programming model requires a lot of expertise, is time consuming, error-prone, and hardwires the execution of a program. In contrast,

relational DBMSs offer a declarative user interface (SQL) and execute automatically optimized query plans. However, RDBMSs are tightly coupled to the relational data model, cannot handle ad-hoc data (well), and have only limited support for user-defined functions.

In this chapter, we propose the Parallelization Contract (PACT) programming model that combines the advantages of relational DBMSs and MapReduce. It eases the implementation of complex analysis tasks while preserving MapReduce's expressiveness. PACT programs are optimized similar to relational queries and compiled into Nephele data flows [203]. This abstraction is similar to parallel execution plans for relational queries and more flexible than the MapReduce execution model. Consequently, PACT programs can often be more efficiently executed than programs implemented with MapReduce. The contributions of this chapter have been published in several articles [9, 10, 11, 23] and were jointly developed with Stephan Ewen.

The remainder of this chapter is organized as follows. Section 3.1 introduces the PACT programming model. Section 3.2 discusses the optimization of PACT programs. We evaluate the PACT programming model in Section 3.3, give an overview of related work in Section 3.4, and summarize this chapter in Section 3.5

3.1 The PACT Programming Model

In this section, we present the PACT programming model as a generalization and extension of the MapReduce programming model. Similar to MapReduce, PACT operators consist of a parallelizable, system-provided, second-order function and a user-defined, first-order function, which process custom data types defined by the user. The system-provided second-order functions are called *Parallelization Contracts (PACTs)* because they give guarantees about how the system will invoke the user-provided first-order function. The term Parallelization Contract motivates the naming of the programming model. The operator set of the PACT programming model includes Map and Reduce as known from MapReduce and three additional binary operators. PACT programs are defined by assembling these operators to acyclic data flows. We start discussing the data model and continue to present the PACT operators. Finally, we discuss the composition of PACT programs.

3.1.1 Data Model

PACT features a generic data model which is centered around the concepts of *data sets* and *records*. It is based on user-defined types in order to support a wide range of use cases. A data set is an unordered list of records and denoted as $D = [t_1, \ldots, t_n]$. A record is an ordered tuple of typed fields $t = \langle v_1, \ldots, v_m \rangle$ with non-null values and field indexes $1, \ldots, m$. Two records $r_1 = \langle v_{1,1}, \ldots, v_{1,n} \rangle, r_2 = \langle v_{2,1}, \ldots, v_{2,m} \rangle$ are equal ($r_1 \equiv r_2$) iff $n = m$ and $\forall i = 1, \ldots, n : \ v_{1,i} = v_{2,i}$. Two data sets D_1, D_2 are equal ($D_1 \equiv D_2$) if there exist two orderings of their records such that $D_1 = [r_{11}, \ldots, r_{1n}]$, $D_2 = [r_{21}, \ldots, r_{2m}]$, $n = m$ and $\forall i = 1, \ldots, n : \ r_{1i} \equiv r_{2i}$.

PACT distinguishes two kinds of field types, *value types* and *key types*. Some PACT operators require the definition of a *key*, which is used to group or associate records. A key can be *atomic* or *composite*, i. e., be defined as a single or multiple record fields. Record fields that are part of a composite key are accessed by the processing system and must implement a specific key interface. The key interface defines methods to compare and hash a key field. These methods are used by the processing system to (re-)organize tuples. Value types are only interpreted by user code and transparently handled by the processing system. Hence, they can be of any arbitrary, user-defined type.

In its original version [23], the PACT programming model was based on MapReduce's key-value pair data model. Later, the data model was generalized and evolved into the here presented record data model. The key-value pair model is a special case of the record model with a single key and a single value field. The record model provides two major benefits over the key-value pair model. First, the record model facilitates more modular data handling and is easier to use. For example, there is no need to define container data types to hold multiple values in a single field as often required by the key-value pair data model. Second, the record model is better suited

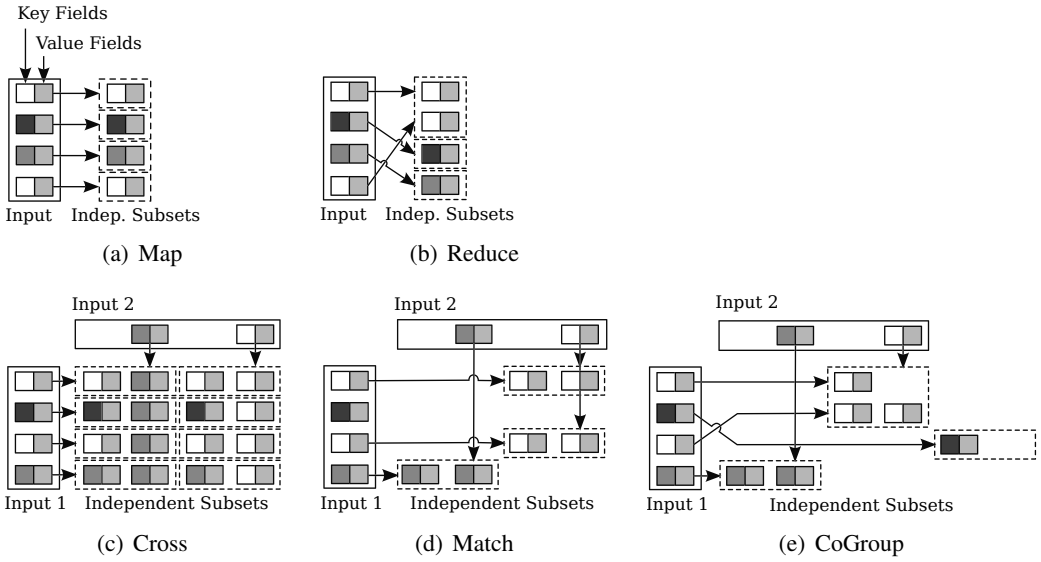


Figure 3.1: System-provided second-order functions

for optimization because it eases the reasoning about data modifications and the preservation of physical data properties.

3.1.2 Operators

A PACT operator is a parallelizable, system-provided, second-order function that wraps an associated, user-defined, first-order function (UDF). PACT operators process and produce data sets of records as defined in the previous section. Conceptually, the second-order function divides the records of its input data set into independently processable subsets and invokes its user-defined, first-order function for each of these subsets. The first-order functions produce one or more records and the output of an operator is the union of the results of all UDF calls. Since all calls of the user-defined function are independent[1], an operator can be executed in parallel. The possible parallelization methods and the potential degree of parallelism depends on the second-order function.

PACT provides five second-order functions that differ in the number of their inputs and how they partition their input data set. While *Map* and *Reduce* operate on a single input, *Cross*, *Match*, and *CoGroup* process the data of two inputs. Map, Cross, and Match operate on one record for each input at a time and are called record-at-a-time (RAT) operators. Reduce and CoGroup process

[1] PACT requires that UDF calls are stateless and have no side effects.

groups of records, which are defined by a grouping key, and are called key-at-a-time (KAT) operators. All user-defined function may produce none, one, or more output records as defined in the previous section. The result of an operator is the (non-duplicate-eliminating) union of all records that are emitted by all of its UDF invocations.

Figure 3.1 shows the second-order functions of the PACT programming model. In the following we briefly describe how these second-order functions partition their input. We assume two input data sets $R = [r_1, \ldots, r_n]$ and $S = [s_1, \ldots, s_m]$.

Map Map has the same semantics as in the MapReduce programming model. In PACT terms, Map is a RAT operator with a single input. It calls its user-defined function exactly once for each record of its input. A Map operator with user-defined function f is formally defined as

$$\texttt{Map}(f, R) = [f(r_1), \ldots, f(r_n)].$$

Figure 3.1(a) visualizes Map's mode of operation.

Reduce Reduce has the same semantics as in the MapReduce programming model. It is a KAT operator with a single input and requires the specification of a key. As described in the data model section, a key is defined as one (atomic) or more (composite) record fields. Reduce organizes all input records into groups with identical key values and calls its UDF for each group providing all records of that group, i. e., all records in one UDF call share the same key value. In the following, we call all records that share the same key a *key group*. A Reduce operator on a key K and with a user-defined function f is defined as

$$\texttt{Reduce}(\mathsf{K}, f, R) = [f(r_1^{\mathsf{k}_1}, \ldots, r_{n_1}^{\mathsf{k}_1}), \ldots, f(r_1^{\mathsf{k}_p}, \ldots, r_{n_p}^{\mathsf{k}_p})],$$

where the domain of values of the key K in R is $\{\mathsf{k}_1, \ldots, \mathsf{k}_p\}$ and the key of a record r^{k} is k. Figure 3.1(b) shows how Reduce applies its UDF to groups of records. Similar to MapReduce, the PACT programming model supports Combiners which are associated with a Reducer.

Cross Cross operates on two inputs and builds the Cartesian product of the records of both inputs. Each pair of records from both inputs is handed once to the user-defined function. A Cross operator with a user-defined function f is defined as

$$\texttt{Cross}(f, R, S) = [f(r_1, s_1), \ldots, f(r_1, s_m), \ldots, f(r_n, s_1), \ldots, f(r_n, s_m)].$$

Figure 3.1(c) visualizes Cross' mode of operation.

Match Match processes two inputs and requires compatible key specification for both inputs, i. e., the data types of both keys must be equivalent. Match builds pairs of records from both inputs whose key values are equal and hands each of these pairs once to its user-defined function. Hence, Match behaves similar to a relational equality inner-join operation but instead of concatenating both record values it calls a UDF. Formally, a Match

operator on two inputs R and S with associated keys K and L and with a user-defined function f is defined as

$$\texttt{Match}(f,\mathsf{K},\mathsf{L},R,S) = [f(r_i,s_j) |\ r_i \in R, s_j \in S, \mathsf{k}_i = \mathsf{l}_j]$$

where k_i (l_j) is the key of record r_i (s_j). Figure 3.1(d) illustrates how Match assembles pairs of records with matching keys.

CoGroup CoGroup operates on two inputs and processes groups of records. It requires the specification of two compatible keys, one for each input, and groups the records of each input by their corresponding key. CoGroup calls its user-defined function with one groups from each input where both groups share the same keys. If there is no matching key in one input for a group in the other input, the UDF is called with that group and an empty group. A CoGroup operator on two inputs R and S with associated keys K and L and with a user-defined function f is defined as

$$\texttt{CoGroup}(f,\mathsf{K},\mathsf{L},R,S) = [f(r_1^{\mathsf{v}_1},\ldots,r_{n_1}^{\mathsf{v}_1},s_1^{\mathsf{v}_1},\ldots,s_{m_1}^{\mathsf{v}_1}),\ldots,f(r_1^{\mathsf{v}_p},\ldots,r_{n_p}^{\mathsf{v}_p},s_1^{\mathsf{v}_p},\ldots,s_{m_p}^{\mathsf{v}_p})],$$

where the combined key domain of distinct K values in R and distinct L values in S is $\{\mathsf{v}_1,\ldots,\mathsf{v}_p\}$ and the key of a record r^{v} (s^{v}) is equal to v. Figure 3.1(e) illustrates CoGroup's mode of operation.

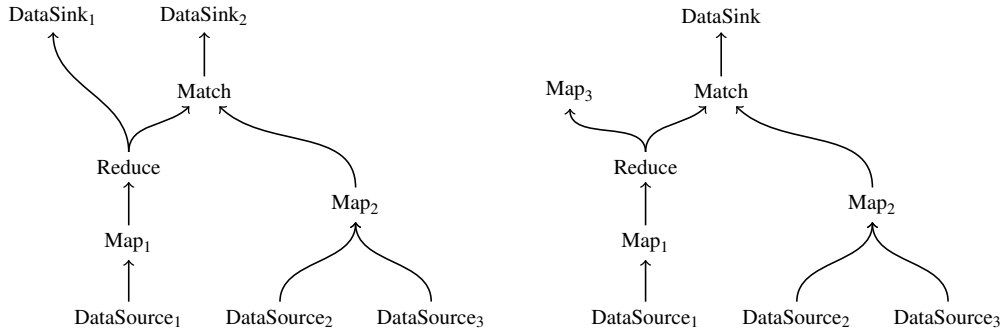
3.1.3 Data Sources and Sinks

The PACT programming model provides five UDF-based operators. In addition to these operators, PACT includes data sources and data sinks. Data sources provide the input data of a program and data sinks consume the result of a program. Data sources and sinks are generic and do not depend on a specific storage system. For example, a data source can read data from a (distributed) file system, query a relational DBMS, or generate data on-the-fly. Similarly, a data sink may write data to files, pipe it to another process, or simply discard it. Since, usually I/O is the bottleneck for data-intensive applications, data sources and sinks can operate in parallel similar to the operators. In order to leverage data parallelism for a data source, the input data must be split into independently processable chunks. Whether and how this is possible depends on the data store and the data itself.

3.1.4 PACT Programs

In the original version [23] of the PACT programming model, a program was defined as a directed acyclic data flow consisting of data source, operators, and data sinks[2]. Conceptually, data

[2]Explicit iteration operators for cyclic data flows were proposed in follow-up work [79, 80].



(a) Valid PACT Program (b) Invalid PACT Program

Figure 3.2: PACT program examples

enters a data flow through data sources, is processed by "flowing" from operator to operator, and finally leaves the data flow through a data sink. The PACT programming model supports programs with multiple data sources and data sinks. All data sources and operators must be transitively connected to a data sink.

Figure 3.2 shows a valid (Figure 3.2(a)) and an invalid (Figure 3.2(b)) PACT program. The right program is not correct due to the Map_3 operator, which does not have a data sink as a transitive successor. The PACT program in Figure 3.2(a) features a few notable details. First, the Reduce operator has two successors, $DataSink_1$ and Match which means that the output of Reduce is duplicated and completely send to both successors. Second, the Map_2 operator has two predecessors while being a unary operator with a single input. In this case, the data of both preceding nodes, $DataSource_2$ and $DataSource_3$, is combined with an implicit (non-duplicate-eliminating) union operator and handed to the Map_3 operator.

```
SELECT
    l_orderkey, o_orderdate, o_shippriority,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer, orders, lineitem
WHERE
    c_mktsegment = :1
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < :2
    AND l_shipdate > :2
GROUP BY
    l_orderkey, o_orderdate, o_shippriority
```

Figure 3.3: TPC-H Query 3 (without final order)

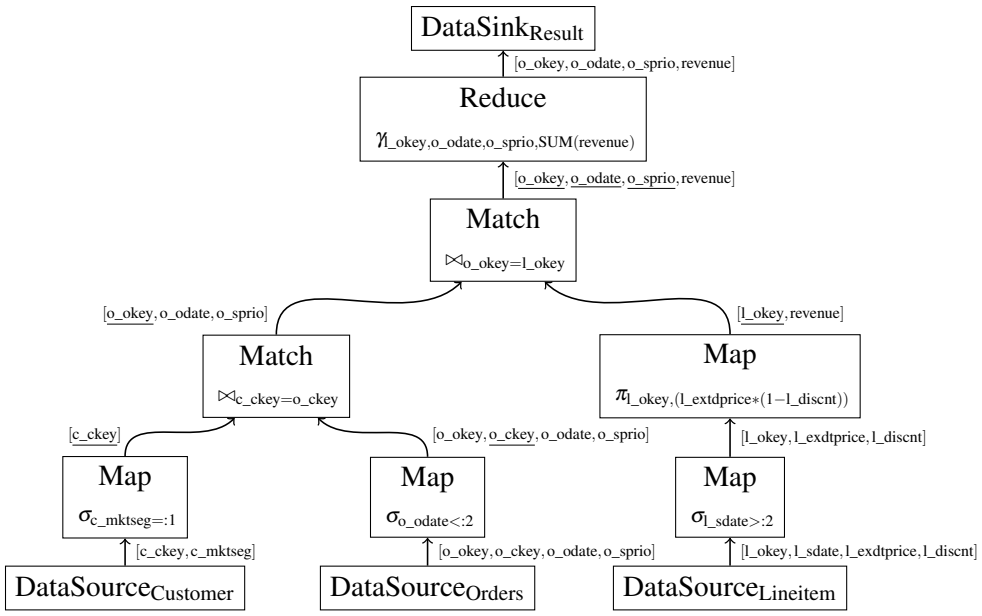


Figure 3.4: TPC-H Query 3 PACT program. The symbols denote the following relational operators: σ selection, π projection, $\bowtie$ join, and γ group-by.

In the following we present and discuss an example PACT program that implements a slightly simplified version of Query 3 from the TPC-H benchmark [200][3]. The equivalent SQL statement for the implemented query is shown in Figure 3.3.

The PACT program implementation is shown in Figure 3.4. The base relations are read by three data sources. In this example, we assume that the data sources perform a full scan of the relations and apply a first projection. In principle, data sources could also perform index look-ups and projections and hence reduce the required I/O operations [70, 71, 172]. Each data source is followed by a Map operator that evaluates a local predicates. The Map operator calls its UDF for each tuple, the UDF evaluates the predicate on its input tuple and forwards it only if it passes. The lineitem input is passed to another Map operator. Its UDF computes for each input tuple the revenue attribute. Both joins of the query are implemented using Match operators with the corresponding join attributes being used as keys. The Match second-order function identifies all pairs of tuples with identical keys and gives them to the user-defined function and the UDF only needs to concatenate two tuples in each call. Finally, the group-by and aggregation of the query is realized using a Reduce Operator with the grouping attributes of the SQL query being set as composite key. The UDF receives all tuples that share the same composite key and computes the sum of all revenue attributes. The result of the Reduce operator is passed to a data sink that emits the result of the query.

[3]The implemented program does not sort the final result.

The PACT programming model extends and generalizes MapReduce in several aspects. Its data model evolves key-value pairs to records of custom types and supports composite key definitions. PACT's operator set includes Map and Reduce and is therefore a superset of MapReduce's. In contrast to the static structure of MapReduce programs, PACT features flexible compositions of operators into acyclic data flows with possibly multiple data sources and sinks.

3.2 The Optimization of PACT Programs

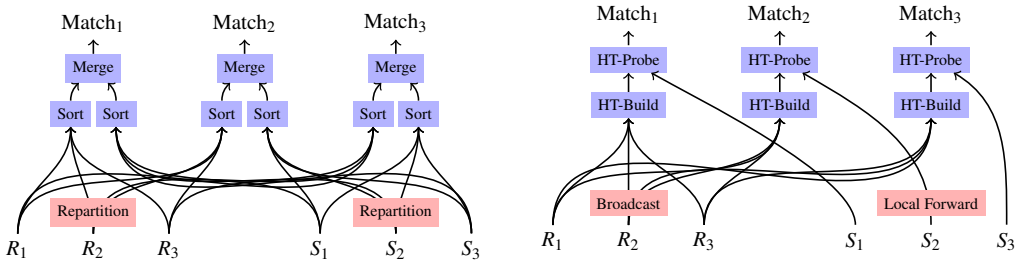
The MapReduce approach tightly integrates a programming and an execution model [64]. Similar to the programming model, the MapReduce execution model is a static, yet scalable, parallel data flow, which is used to execute all MapReduce programs[4]. The PACT programming model offers more flexibility than the MapReduce programming model. PACT programs are acyclic data flows composed from operators, which consist of a system-provided, second-order function and a user-defined, first-order function. For execution, a PACT program is translated into a parallel data flow that is processed by a general parallel data flow system as described in Section 2.1.4. In contrast to MapReduce, this translation exhibits several degrees of freedom which can lead to different data flows with vastly varying execution performance [10, 23]. Hence, the compilation of PACT programs into parallel data flows bears similarity with the translation of SQL queries into execution plans as performed by relational DBMSs. However, there are also a few notable differences to relational optimization. In this section, we discuss the cost-based optimization and translation of PACT programs into parallel data flows.

3.2.1 Execution Strategies

PACT programs are compiled into and executed as parallel data flows. As described in Section 2.1.4, operators of parallel data flows process on partitioned data. Each data partition is independently processed by a parallel instance of an operator. This mode of operation follows the principle of data parallelism. A PACT program can be often executed by multiple different but semantically equivalent data flows. The degrees of freedom originate from the fact that PACT's system-provided, second-order functions exhibit a declarative character. Similar to a relational join, PACT's Match operator does not imply how the execution system brings matching records together. Instead, the program compiler can choose from different physical execution strategies to perform this operation. The PACT optimizer distinguishes between two types of execution strategies, *local strategies* and *ship strategies*. A ship strategy defines how the data is partitioned among the parallel task instances of an operator, i. e., which data a parallel task instance processes. A local strategy defines how the parallel task instances of an operator process the data of their individual partitions.

Figure 3.5 shows two alternative execution strategies for a Match operator, which is semantically close to a relational inner equi-join. Red boxes highlight ship strategies, blue boxes local strategies. The left alternative (Figure 3.5(a)) shows a repartition sort-merge execution strategy. Both inputs, R and S, are repartitioned on their respective Match key, i. e., all records with the same key are sent the to the same parallel instance of the Match operator. Each parallel instance of the Match operator individually sorts the records of both inputs on their keys and subsequently

[4] In database terminology, all MapReduce programs are executed with the same execution plan.



(a) Repartition Sort-Merge Match (b) Broadcast (Hybrid-)Hash Match

Figure 3.5: Alternative execution strategies for Match operators

performs a merge join to identify record pairs with matching keys. Finally, the UDF of the Match operator is called for each record pair with matching keys. The alternative plan shown in Figure 3.5(b) uses a different ship strategies for each input. The first input R is broadcasted to each parallel instance of the Match operator, i. e., the data is fully replicated on each instance. Each parallel instance builds a hash-table from the broadcasted data indexed by R's Match key. The data of the second input is locally forwarded to the corresponding Match instance, i. e., it is not shipped over a network connection, and probes the hash-table with the key of each record to identify pairs of records with matching keys. For each record pair, the UDF of the Match operator is called. There are further alternatives to execute a Match operator. For example, repartition ship strategies can be combined with a hash-join local strategy or the build- and the probe-side of a hash-join or the broadcasted and the forwarded inputs can be switched.

3.2.2 Interesting Properties

Local and ship strategies reorganize the data and establish certain properties on distributed data sets. Such properties include sorting, grouping, and different types of partitioning. Conceptually, operators require that their input data holds certain properties. For example a Reduce operator requires that its input data is grouped by key. In order to establish a distributed data set with the required grouping property, records need to be grouped by key across and within partitions, i. e., all records that share the same key must be present and grouped together in the same partition. Repartition ship strategies (hash- or range-partitioning) can be used to group records across partitions. Grouping within a partition can be achieved by sorting the records or applying hashing techniques.

An important aspect of data flow optimization are operators that produce data sets with properties, which can be reused by a subsequent operator. If for example, an operator produces a data set that is hash-partitioned in a suitable way for a following Reduce operator, the data can be locally forwarded instead of using an expensive repartitioning strategy that sends all data

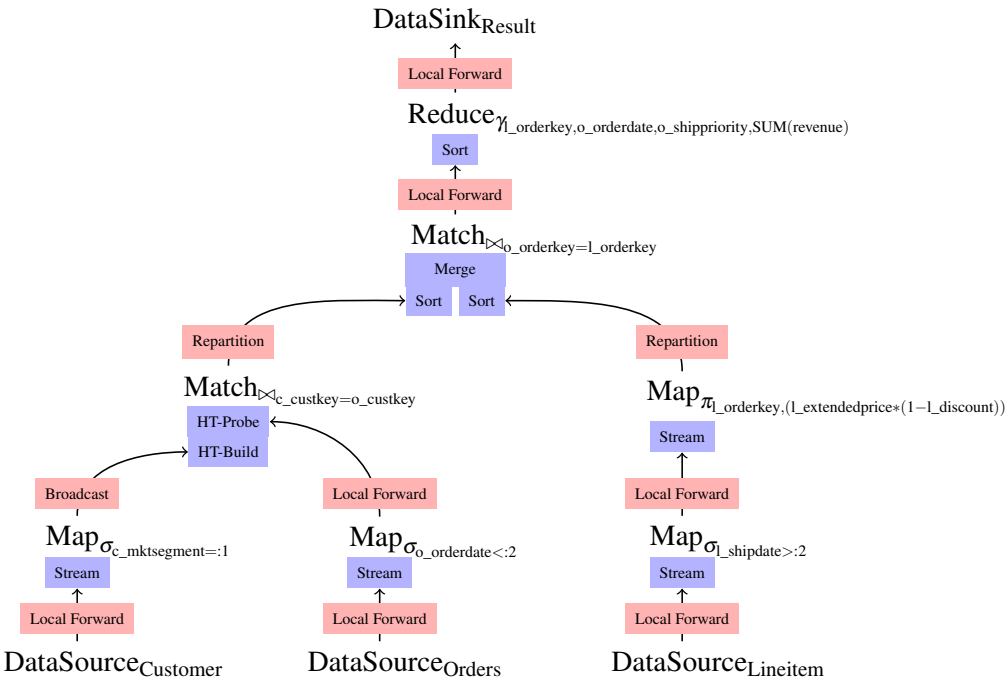


Figure 3.6: TPC-H Query 3 data flow

over the network. Optimizers that consider these opportunities can produce significantly more efficient data flows. More optimization opportunities arise if reusable data properties are not only leveraged when found, but when optimizers also explicitly enumerate plans that provide such opportunities. Plan enumeration techniques that explicitly generate and memorize plans, which produce so-called interesting properties are well-known from relational query optimization [101, 186, 213]. These techniques usually work in two steps. First, they enumerate all properties that can be reused by an operator and are hence interesting. Second, the optimizer generates plans where the input data of the operator exhibits interesting properties and memorizes them even if they are more expensive than equivalent plans but less expensive plans.

Figure 3.6 shows a candidate data flow to execute the TPC-H Query 3 example PACT program presented in the previous Section 3.1. Observe that the aggregating Reduce operator receives its input data from a local-forward ship strategy, i. e., its input data is not transferred over the network. Instead, Reduce reuses the partitioning that was established for the previous Match operator. This is possible, because Reduce's key $(\text{l\_orderkey}, \text{o\_orderdate}, \text{o\_shippriority})$ is a superset of Match's key (l_orderkey)[5]. Reasoning about interesting properties can lead to several optimization opportunities especially in presence of composite keys. For examples, data can be

[5]The optimizer can even decide to sort the input of the Match operator on the full key of Reduce and reuse the sorting for the Reduce if it knows that the Match operator preserves the sorting.

hash-partitioned only on a subset of a composite key to ensure these properties are reusable for a subsequent operator. It should be noted that such optimizations require distribution statistics on the partition keys as they can also worsen data skew. Zhou et al. formalized the interesting property reasoning of the Scope optimizer [213].

However, there is an additional requirement that needs to be met to enable effective interesting property optimizations for PACT programs. In contrast to relational operators, PACT operators do not have well-defined semantics due to their user-defined function. UDFs can arbitrarily modify their input data and also change the key fields of a record. Hence, the optimizer cannot assume that the output data of an operator has the same properties as the data that was passed into its UDFs. Instead, the optimizer must assume that all properties are eliminated by the UDF.

To overcome this problem, the PACT programming model offers annotations for UDFs to reveal some of their semantics to the optimizer. In the original description of the programming model [23] that used a key-value pair data model, these annotations where called *Output Contracts* and could specify whether the key field was unmodified or extended by a UDF. Given such an annotation, the optimizer could infer that properties of the input data, such as partitioning or sorting, were preserved by a user-defined function. These output contract annotations changed in the course of the data model evolution from key-value pairs to records. In the current version, UDF annotations can specify which fields of a record are not modified by a UDF. This allows more fine-grained inference of preserved properties in case of composite keys.

3.2.3 Cost Estimation

A PACT program can be compiled into multiple different but semantically equivalent parallel data flows. These data flows differ in execution strategies and consequently in execution times as well. Relational database systems are in a similar situation when choosing the execution plan for a SQL query and commonly employ cost-based query optimizers to identify the execution plan with the least estimated cost. Cost estimates for plans are computed by a cost model as a function of input data (cardinality and size) and available resources (memory budget, I/O rate, CPU utilization).

In principle, the PACT optimizer follows a similar approach as relational database optimizers. However, there are a few notable differences. While cost models of relational database systems are quite advanced and fine-tuned [105, 107], PACT follows a more coarse-grained approach. This is mainly due to lack of accurate information. Relational DBMS are able to leverage different kinds of information for cost estimations, including schema information and statistics, such as base cardinalities and value distributions. In order to estimate the cost of an operator, the size and cardinality of all its inputs must be estimated. This is achieved using base statistics and reasoning about the semantics of all logical operators that are applied to compute the operator's

input [186]. However, cost estimation and in particular the size estimation of intermediate results are hard problems and still a major challenge for modern database optimizers [131, 164].

In the context of the PACT programming model, cost estimation is even harder. PACT does not feature a dedicated data store but processes potentially unstructured ad-hoc data for which no or only very limited statistical information is available. Furthermore, estimating the cardinality of intermediate results is significantly harder due to the unknown semantics of the operators' UDFs. Given the limited information that can be leveraged for size estimations, the PACT optimizer would not benefit from a too detailed cost model. Instead, it follows a pragmatic approach and tries to extract basic statistics from data sources. For example for file inputs, the optimizer draws a few sample records and estimates the input cardinality as the ratio of total file size and average record width of the samples. Having obtained basic statistics for the input of the program, the optimizer needs to estimate the size of intermediate results, i. e., the size and cardinality of an operator's output for a given input. In general, this is not possible because neither the schema of the input or output data nor the semantics of the operators' UDFs are known. PACT's approach to overcome this problem is to offer compiler hints that allow to specify the selectivity of a UDF, the average size of output records, and the average number of records for unique combinations of fields. The optimizer computes the result cardinality of a PACT operator by estimating the number of UDF calls and combining this information with an available compiler hint. Given a Map operator, the optimizer knows that the UDF is called once for each incoming record and multiplies this with the selectivity hint of the Map UDF. In principle, compiler hints can be obtained in different ways. Users can specify hints when implementing a PACT program or compilers that translate higher-level languages or programming abstractions such as Pig, Hive, or Cascading into PACT programs can provide them due to their richer semantics. Another option is to employ runtime monitoring or static code analysis techniques to obtain compiler hints. If no compiler hints are available, the optimizer marks estimates as unknown and falls back to conservative execution strategies such as repartition shipping strategies instead of broadcast-forward strategies.

The cost model of the PACT optimizer aims to minimize the amount of data shipped over the network and the amount of data that is written to external storage, i. e., hard disk. The model computes those two estimates but does not combine them into a joint cost metric. Instead, it primarily evaluates the amount of network-transferred data and secondarily checks for estimated disk I/O. The rational for this mode of operation is that network transfer is significantly more expensive than disk I/O. Since shipping and local strategies can be often freely combined, both metrics are useful.

3.2.4 Differences to Relational Optimization

The PACT optimizer leverages techniques which are known from relational optimization, such as its alternative execution strategies, interesting property reasoning, and cost-based plan choices.

However, there are also several aspects that differ from relational optimization.

The shape of PACT programs can differ from relational queries. SQL queries compute a single result and have a tree structure, i. e., the result of each operator is forwarded to exactly one successor. PACT programs may have more than one sink, i. e., can compute more than one result. Furthermore, a PACT operator may forward its output to multiple successors such that data flows can branch and join again. These differences requires special plan enumeration techniques.

Also the job abstraction that is given to the optimizer for optimization differs. Relational database systems parse a SQL query and construct a logical operator tree, which is handed to the optimizer. This representation is based on relational algebra operators with well-defined semantics. Using a set of well-known logical transformation rules, the optimizer is able to reorder or replace logical operators. In contrast, PACT programs are concrete data flows consisting of UDF-based operators with only partially known semantics. This lack of knowledge impedes reasoning about the preservation of interesting properties, accurate size estimates, and the reordering of PACT operators. In its initial version, we addressed the first two aspects by providing annotations and optimizer hints to reveal some information to the optimizer[6]. The challenge of reordering UDF-based operators is addressed in Chapter 4.

As discussed in the previous section, cost-based optimizers rely on the accuracy of cost estimation. PACT's lack of operator semantics does severely affect its ability to precisely estimate the size of intermediate results. However, lack of size estimation accuracy is also a challenge for relational database systems. Chapter 5 analyzes the effects of imprecise optimizer estimates.

[6] PACT can serve as a compilation target for higher-level languages [117, 140]. Annotations and compiler hints enable a language compiler to inject additional semantics into a PACT program.

3.3 Evaluation

The PACT programming model advances MapReduce in mainly two dimensions. First, it eases the definition of advanced data analysis programs while not giving up the approach of a general programming model. Second, it offers better performance and efficiency than MapReduce due to its optimization and compilation to parallel data flows. In this section, we compare MapReduce and PACT along both dimensions with the help of a few example programs.

3.3.1 Ease of Use

An important aspect of any programming model is its usability, i. e., the effort that is required to solve a problem. It is not easy to objectively evaluate the usability of a software or programming model. A common approach are user studies, which can be biased due to the skill level of the probates and the problems under consideration. The fact that PACT is a superset of MapReduce eases their comparison. In this section, we present a few common data analysis tasks that are not straightforward to realize with MapReduce and show how PACT improves these. Finally, we show PACTs superiority by comparing the definitions of two analysis programs in MapReduce and PACT. Several aspects of this comparison have been discussed in a previous publication [11].

The MapReduce programming model is a static template for data processing programs. Data is read, processed by a user-defined Map function, grouped, processed by a user-defined Reduce function, and finally emitted. There are several common data processing tasks that do not nicely fit this template. Among these tasks are data analysis programs that analyze more than one data set. The MapReduce programming model features only a single logical input. Hadoop's [109] implementations of the MapReduce model offers abstractions to read multiple data sets. One solution is to put a file into the so-called distributed cache before starting a job and let each Mapper read the file from the cache. This technique broadcasts the data of the file to each Mapper but it is only applicable for small files. Another solution is to register multiple input formats and Map functions. Each input is processed by an individual input format and Map function. The data of both inputs is handed together to the same Reduce function. In order to track which key-value pair originated from each input the Map functions need to inject a lineage flag into the key-value pairs. In contrast, the PACT programming model supports multiple data sources that can be unioned or combined using one of the three binary operators, Cross, Match, and CoGroup.

The key-value pair model is another limitation of the MapReduce programming model. It forces programmers to squeeze all data items into two data types, a key and a value. This results in struct-like container data types to hold unrelated data items. For each unique combination of data items, such as a composite key, a new container type needs to be defined. As a consequence,

the key-value pair model increases development overhead and hampers the reasoning about programs. PACT's record model facilitates the reuse of data types and supports the definition of composite keys.

Even for moderately advanced data analysis tasks it is common that the logic of an application cannot be expressed as a single MapReduce program and needs to be distributed over multiple MapReduce programs, which depend on each other's output. Such analysis programs are executed as workflows of chained MapReduce programs. There are several drawbacks to this approach. First, analysis programs are cumbersome to implement and maintain if the program logic is distributed over multiple MapReduce programs. Second, the execution of the individual MapReduce programs needs to be orchestrated[7]. Finally, executing analysis programs as workflows of MapReduce programs has considerable performance implications. MapReduce programs can exchange data only by reading and writing to stable storage systems, such as distributed file systems[8]. Also the execution of each MapReduce program comes with high overhead costs due to task scheduling and mandatory data shuffling and sorting. In contrast, the PACT programming model defines data processing programs as arbitrarily complex data flows. Hence, there is no need for external job orchestration, reading and writing of intermediate results from and to stable storage, and reduced scheduling overhead. Instead, the data flow model offers potential optimization benefits.

The more declarative nature of PACT is another feature that distinguishes it from MapReduce. Declarativity is not only an enabler for automatic optimization, it also influences the usability of a programming model since it hides unnecessary complexity from users. Using declarative concepts, a user specifies what should be done and does not need to bother how it is achieved. Instead, this task is delegated to an optimizer and the execution engine. An example for this is PACT's Match operator. As mentioned before, Match resembles an inner equi-join. There are two common strategies to implement an equi-join in MapReduce. The Map-side join leverages the distributed cache to broadcast the (sufficiently) smaller join input to each Mapper and builds a hash table from it, indexed by its join attributes. Subsequently, the Map function processes the larger input by probing the hash table for each key-value pair and constructs and emits a joined output key-value pair for each match. The alternative Reduce-side join reads both inputs using a special input format for multiple inputs. The Map function attaches a lineage tag to each key-value pair. The data of both inputs is unioned and partitioned and sorted by join key. Finally, the Reduce function builds joined key-value pairs from all key-value pairs with matching key originating from different inputs. The MapReduce implementation of an analysis program that includes a join is fixed to the chosen join implementation. However, the best join strategy depends on the size of the inputs and the program's degree of execution parallelism, among other things. The PACT programming model offers the Match operator and lets an optimizer choose which join strategy to use depending on the programs input size and execution environment.

[7] There are schedulers and optimizers such as Oozie [170] and Stubby [151] for MapReduce workflows.
[8] Distributed main memory file systems, such as Tachyon [112, 196], reduce these costs.

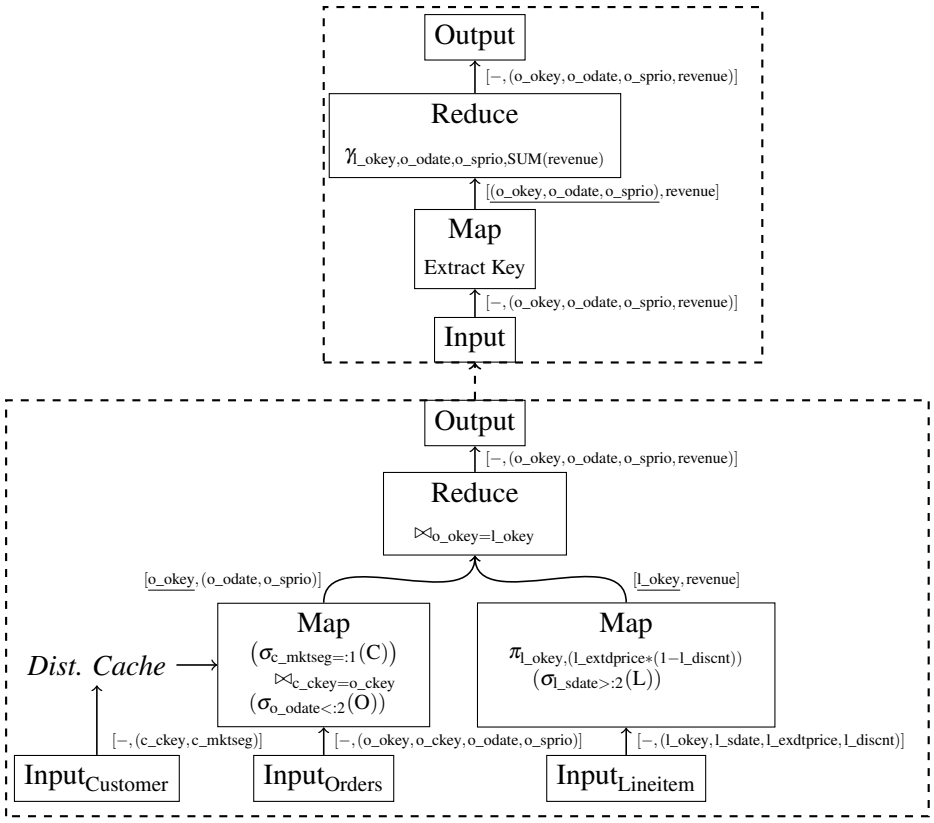


Figure 3.7: TPC-H Query 3 MapReduce workflow

In fact, the Reduce-side join and the Map-side join follow the same strategy as the execution strategies shown in Figures 3.5(a) and Figure 3.5(b), respectively.

In the following, we compare the MapReduce and PACT implementations of two data analysis tasks. The first analysis program is the TPC-H query that we used as an example in the previous sections. The second program is an algorithm that identifies all triples of vertices in a graph which are connected with each other. This triangle enumeration task is commonly used as a preprocessing step to identify densely connected subgraphs, e. g., for social network analysis.

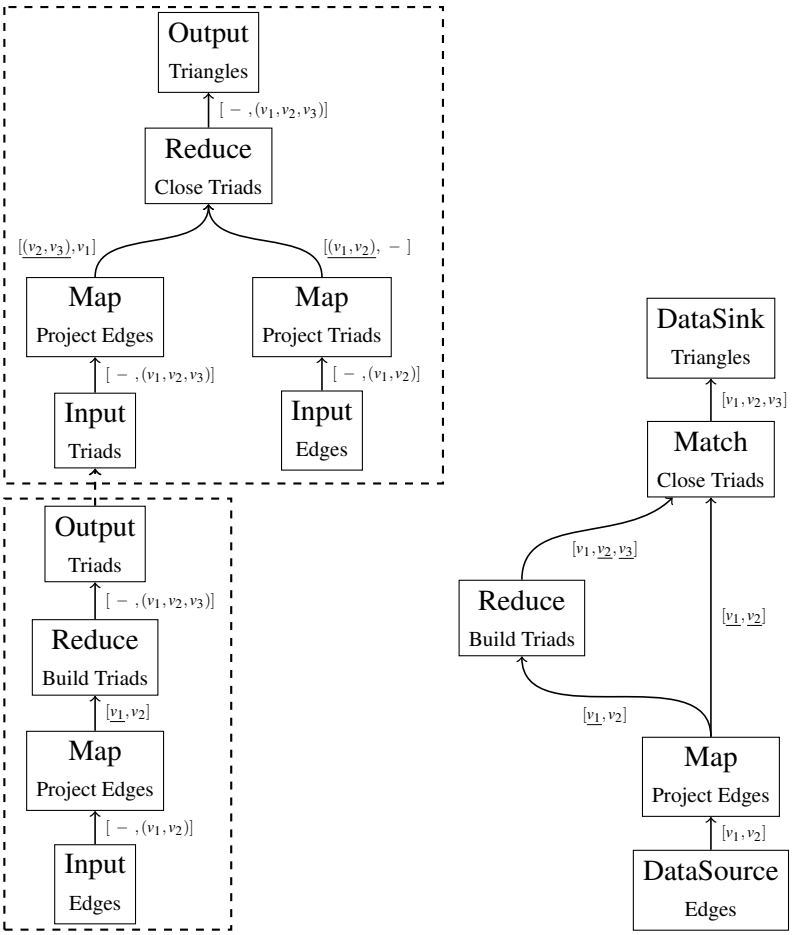
Relational Query We use TPC-H query 3 as a running example previously in this chapter. Figure 3.3 shows the SQL statement and Figure 3.4 shows an implementation using the PACT programming model. Because MapReduce does not abstract logical programs from physical execution, there are several ways to implement this query in MapReduce. Figure 3.7 shows a MapReduce implementation, which is moderately optimized for performance and not the most

straightforward solution[9]. It is based on two MapReduce jobs, of which the first one is rather complex. The first job reads the `orders` and `lineitem` relations using a multi-input input format and processes both relations with individual Map functions. At start-up time, the `orders` Map function reads the `customer` relations from Hadoop's Distributed Cache, which is used for broadcasting small data sets, and builds an in-memory hash-table. Then it starts processing the `orders` relation, applies a filter predicate, and joins `orders` with `customer` by probing the hash-table. This join strategy is known as Map-side join. The `lineitem` Map function also applies a filter predicate and computes the `revenue` attribute. Both Map functions return key-value pairs with the key being set to the `o_okey` or `l_okey` attribute, which are the join attributes for the following join. The following Reduce function joins results of both Map functions. For each unique key (`o_okey` or `l_okey`), the Reduce function is called once. The function collects all input pairs and separates them into two groups depending on their origin. Finally, for each combination of pairs of both groups a key-value pair is emitted and written to a stable storage, such as HDFS. The second job reads the data emitted by the first job and applies a Map function that emits for each input record a key-value pair with a key that contains the `o_okey`, `o_odate`, and `o_sprio` attributes. The following Reduce function simply sums the `revenue` attribute in each group and emits one key-value pair for each group, which is emitted by the final output format.

Looking at the presented MapReduce implementation, we see that a large fraction of the program logic is squeezed into the Map function that joins the `customer` and `orders` relation while the Map function of the second job does only set the correct key. Such patterns are quite common when implementing workflows of MapReduce jobs. Since each job comes with a high execution overhead for reading, writing, shuffling, and sorting the data, and scheduling all task instances, minimizing the number of jobs is an important optimization. This naturally leads to UDFs with highly intertwined program logic. The resulting source code is hard to maintain and barely reusable. In contrast, the PACT programming model encourages a more modular programming style as shown in the PACT variant of the TPC-H query in Figure 3.4. Here, each function serves a single purpose, such as a filter, projection, or join.

Another issue with the MapReduce model are hard-coded execution strategies. For example, the presented query implementation relies on the fact that the filtered `customer` relation is small enough to fit into an in-memory hash table. Also distributing the `customer` data via the Distributed Cache becomes less efficient if the relation grows or the execution parallelism of the job increases. As a result, MapReduce implementations are often tailored towards specific input sizes, size ratios, or degrees of parallelism and become less efficient if these change. Moreover, manual optimization of data flows is also a non-trivial task and requires experience and programming skills. The PACT programming model addresses this issue with an optimizer that automatically chooses the execution strategies for a PACT programs depending on the input size and degree of parallelism.

[9] There might be even more efficient implementations.



(a) MapReduce Program (b) PACT Program

Figure 3.8: Triangle Enumeration programs

Triangle Enumeration Our second example for comparing the MapReduce and PACT programming models is an algorithm to enumerate triangles, i. e., three-edge cycles, in undirected graphs. The basic algorithm and a MapReduce implementation were presented by Cohon [57]. We adapted this algorithm for the PACT programming model. Figure 3.8 shows both variants.

The algorithm takes a list of edges as input. An edge is represented as a pair of vertices. The edges are grouped by their lexicographically smaller vertex. Within each group, the algorithm enumerates all pairs of edges and builds so-called triads. A triad consists of two edges that share a common vertex (the grouping vertex) and is represented as a triple of vertices. Finally, the triads for which no closing third edge in the original edge set exists are removed.

The MapReduce implementation (Figure 3.8(a)) consists of two MapReduce programs. The first program, reads the input edges and uses the Map function to generate key-value pairs where the lexicographically smaller vertex ID is set as key and the other vertex ID is set as value. The following Reduce function builds all combinations of edges within a group and emits for each pair of edges a triad. Finally, the first program writes all triads to a stable storage. The second program is invoked after the first program finished. It reads two inputs. The first input are the previously computed triads and the second input is the original set of input edges. The Map function for the triads generates key-value pairs with the two vertices, which are not the common vertex of the triad's edges, set as key and the common vertex of the triad set as value. For the edge input, the whole edge is interpreted as key. In both cases, the key vertices are ordered by ID. Triads and edges are grouped by key and the Reduce function checks for the presence of an original input edge. If an edge is present, it constitutes the missing edge and all triads of that group are emitted as triangles. If no edge is present in a group, the group's triads are discarded.

The PACT program is shown in Figure 3.8(b). It starts reading the input edges and applies a Map operator to lexicographically order the vertex IDs of each edge. The result is given to a Reduce function that treats the first attribute as key and builds triads by combining all edges of a group. The result of Reduce and the Map operator are forwarded into a Match operator. The key of the Mapper's edges are both vertex IDs, the key of the Reducer's output are the triad's non-common vertex IDs. The Match operator calls the UDF only for matching pairs of records of both inputs. Hence, all triads without a matching edge are automatically discarded by the system's Match operator. The UDF simply emits all triads.

When comparing both implementations, the most obvious difference is that two MapReduce programs are required, whereas the PACT implementation consists of a single data flow program. Looking closer, we see that the first part of the PACT program is almost identical to the first MapReduce program. However, MapReduce realizes the following join between triads and edges with a Reduce-side join strategy, which requires a handcrafted join implementation as an additional MapReduce program. In contrast, the PACT program leverages the Match operator to perform the join using system-internal operations. The Match UDF implementation is trivial and only needs to forward its input triad. One effect of having two MapReduce programs is that the edge input needs to be read twice instead of once as in the PACT implementation. For the MapReduce program, a large amount of implementation effort is spent on assembling the correct grouping and join keys. This needs to be manually implemented in all three Map functions. In addition, custom key and value data types are required to hold edges, and triads. The PACT program processes data as records and the composite keys are defined by selecting fields of the input record.

3.3.2 Performance

An important property of any data processing system is its performance and processing efficiency. While the MapReduce execution model provides good scalability and fault-tolerance, it has been criticized for its inefficiency and low performance [173, 194]. One can argue that MapReduce is, with respect to efficiency, not a good fit for some of the use cases it is applied to. On the other hand MapReduce is widely adopted as indicated by the popularity of higher-level programming abstraction on top of Hadoop [44, 169, 198]. In this section, we compare the performance of MapReduce and PACT programs by discussing the results of an experimental evaluation that was published in the VLDB Journal [10]. Specifically, we look at the implementations of three analysis tasks, 1) Word Count, 2) TPC-H query 3, and 3) Triangle Enumeration.

The evaluation compares Stratosphere [195], our implementation of the PACT programming model, and version 1.0.4 of Apache Hadoop [109] and was carried out on a cluster of 25 compute nodes and one dedicated master node. Each compute node was provided with two AMD Opteron 6128 CPUs with 8 cores each, 32 GB RAM, a Gigabit Ethernet adapter, and four SATA hard disks. For the evaluation the number of compute nodes was varied from 5 to 25 with 8 parallel tasks running on each worker node yielding a total parallelism of 40 to 200. For all benchmarks, the amount of data is scaled proportionally with the number of worker nodes. We refer to the original publication for details [10].

Since the execution engines of Stratosphere and Hadoop differ, this section does not compare the performance of the PACT and MapReduce programming models in isolation. Certain performance differences cannot be exclusively accounted to the programming model but are an artifact of the interplay of programming model and execution engine. For instance, Stratosphere's execution engine features pipelined data shuffles, i. e., records are shipped to succeeding operators as soon as they are produced. In contrast, Hadoop employs batched shuffles by collecting records on the sender side and sending them to the receiver them in large batches. Due to pipelined shuffles, consecutive operators can concurrently process data in Stratosphere. Further differences are the implementations and available types of internal operators such as sort algorithms and hash tables.

Word Count Word Count is the classic example to introduce and explain MapReduce [64]. The task is to compute a histogram of words, i. e., count the occurrences of each word in a text corpus. This problem maps naturally to a single MapReduce program. The Map function receives lines of text and splits it into key-value pairs with a word being the key and a count being the value. The count is initially set to 1. The Reduce (and Combine) function groups the pairs by word and compute the sum of counts. The PACT implementation is conceptually equivalent to the MapReduce solution and executed using the same strategies as the MapReduce execution model. Hence, this benchmark compares the efficiency of the execution engines and

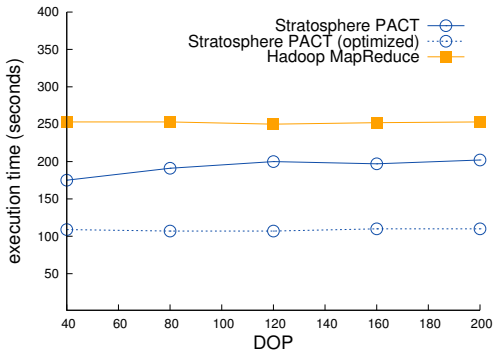


Figure 3.9: Execution time of MapReduce and PACT Word Count programs for varying degrees of parallelism (DOP) [10].

gives a baseline for the following performance comparisons. It does not compare the efficiency of the programming models.

Figure 3.9 shows the execution time of a Hadoop MapReduce and a PACT Word Count program for varying parallelism and amounts of data. Both systems scale linearly with increasing numbers of worker nodes. The difference in execution time can be explained with Stratosphere's pipelined communication model, a different implementation of the Combiner, and its sort implementation, which works (partially) on binary data instead of deserialized objects [10]. The "Stratosphere PACT (optimized)" experiment used a more efficient tokenizer to improve the performance of the Map function. This tokenizer could also be used in a Hadoop MapReduce job [10].

This benchmark shows that the performance of Stratosphere's processing engine is competitive to Hadoop. Since both systems use the same execution strategy to execute the Word Count program, Stratosphere's superior performance cannot be explained by the PACT programming model. Instead it is the result of a more efficient runtime system.

TPC-H Query 3 In the previous section we compared two possible implementations of the running TPC-H query 3 example in MapReduce and PACT. The experimental evaluation of the Stratosphere journal paper [10] also contains a performance comparison for this query. However, instead of using a handcrafted MapReduce program such as the one shown in Figure 3.7, the benchmark was performed with Apache Hive [124, 198] (version 0.10.0), a SQL compilation layer on top of Hadoop. Hive includes a rule-based optimizer that aims to minimize the number of MapReduce programs.

Figure 3.10 shows the performance of the TPC-H query 3 as shown in Figure 3.3 executed as a PACT program on Stratosphere and ran as a Hive query on Hadoop MapReduce. The execution

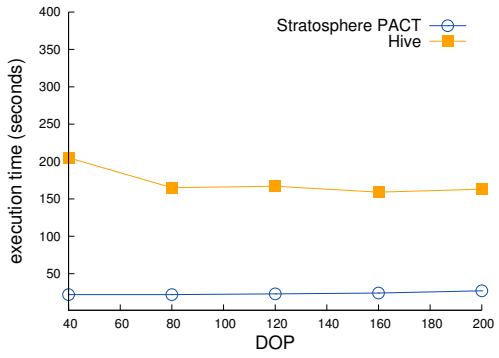
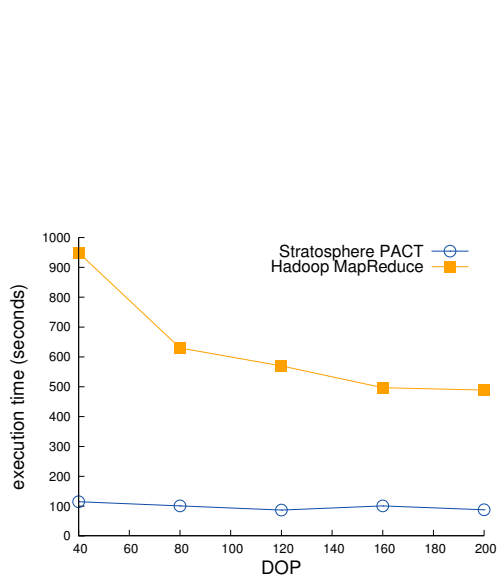


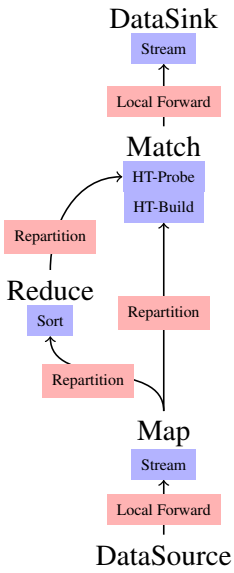
Figure 3.10: Execution time of MapReduce and PACT TPC-H Query 3 programs for varying degrees of parallelism (DOP) [10].

time of the query with Hive on MapReduce is significantly higher than the execution time of the PACT implementation. This is due to the fragmentation into multiple MapReduce jobs, which requires to pass the result of a job to the next one via a distributed file system, the mandatory shuffle and sort strategies of the MapReduce execution model, and the additional scheduling overhead. In contrast, the PACT program is executed as a single data flow. The `lineitem` input, which accounts for more than 70% of the input size, is joined using a broadcast hash-partition join. Since the hash table and the sort for the final aggregation fit completely into memory, no intermediate disk I/O is performed.

Triangle Enumeration The experimental evaluation in [10] also compares the performance of hand-crafted MapReduce and PACT programs for the triangle enumeration task, which was presented in the previous section. Figure 3.11(a) shows that the PACT program significantly outperforms the MapReduce implementation. The PACT optimizer picks a plan that minimizes the intermediate I/O operations. The plan is shown in Figure 3.11(b). The most important aspect of this plan is how the result of the Reduce operator is handled. The Reduce function builds all combinations of edges that have the same grouping vertex. For n edges that share the same vertex, the Reduce function generates $\frac{n*(n-1)}{2}$ output triads. In the MapReduce program, this asymptotically quadratic intermediate result is written to a distributed file system, read again by the following job, repartitioned, sorted using external storage, and finally processed and emitted by the next Reduce function. In contrast, the PACT program repartitions the triads in a pipelined fashion (without materialization) and identifies valid triangles by probing hash tables that were built using the original edges. If these hash tables fit into memory, the large intermediate triad data set is not materialized and completely processed on-the-fly.



(a) Execution time of MapReduce and PACT Triangle Enumeration Programs for varying degrees of parallelism (DOP) [10].



(b) Data Flow of PACT Triangle Enumeration Program.

Figure 3.11: Triangle Enumeration

3.4 Related Work

In this section, we briefly discuss how the PACT programming model and its optimization techniques relate to other techniques, approaches, and systems.

As previously discussed, the PACT programming model is a generalization of the MapReduce programming model [64]. It borrows the concept of parallelizable system-provided second-order functions and user-defined first-order functions. Compared to MapReduce, PACT offers more parallelization primitives, supports the composition of arbitrary complex acyclic data flows, and features a record data model. Moreover, it is not tightly coupled with an execution model. In contrast, a cost-based optimizer compiles PACT programs into parallel data flows.

With the growing popularity of MapReduce, it became clear that its static programming model is not a good fit for several real-world problems. Especially, more advanced data analysis tasks can only be implemented in a clumsy and cumbersome way using the MapReduce programming model, as shown by our evaluation in the previous section. Driven by this insight, several efforts started to improve its usability from a programmer's point of view. Among these approaches are higher-level languages such as Apache Pig [169, 175], Apache Hive [124, 198], and Jaql [28],

and programming APIs such as Cascading [44]. These programming abstractions address some of the shortcomings of the MapReduce programming model that motivated and influenced the design of the PACT programming model. From an API point of view, Cascading is similar to PACT. It offers similar parallelizable operators that run custom user code including CoGroup and Join. The operators can be assembled into complex data flows. However, these Cascading flows are translated into workflows of MapReduce programs. Languages such as Pig, Hive, and Jaql offer a more higher-level abstraction than PACT to specify data analysis tasks. In their original versions these languages were also compiled into workflows of MapReduce programs. Compiling programs or queries to workflows of MapReduce programs offers only limited optimization opportunities due to the inflexible MapReduce's execution model. Moreover, the sequential execution of several MapReduce programs comes at high overhead costs as discussed before. In contrast, the PACT optimizer is able to reason about interesting data properties and compiles PACT programs into efficient parallel data flows. Recently, Hive, Pig, and Cascading added support for compiling to Apache Tez [197], a parallel data flow execution system. This step improves their execution efficiency, but also calls for an optimizer such as PACT's. In fact, the PACT programming model itself can also serve as a compilation target for higher-level languages such as Pig [140], Hive, Jaql, and Cascading.

Apache Spark [190, 207, 208] is a system for parallel data analysis. Its programming model is based on resilient distributed datasets (RDDs), on which transformations and actions are applied. Spark's transformations are similar to PACT's operators and include Map, Reduce, and Join operators. Similar to PACT, Spark programs are compiled into parallel data flows and their execution is triggered by actions. However, the compilation step does not involve advanced optimizations such as interesting property reasoning or cost-based plan choices. As a system, Spark put more focus on in-memory processing and low task scheduling overhead.

Another field of related work are parallel relational database systems [35, 67, 68, 96, 183, 210]. PACT's optimizer borrows concepts from relational optimizers including cost-based optimization [186] and interesting property reasoning [101, 186]. As pointed out in Section 3.2.4, there are also notable differences to relational optimization such as PACT's DAG-shaped programs and its focus on user-defined functions. Nonetheless, a parallel data flow which is produced by the PACT optimizer is similar to an execution plan of a parallel, shared-nothing, relational database system. Both are built from the same physical execution strategies such as hybrid-hash join and sort-based grouping strategies for local processing and repartitioning and broadcasting strategies for data exchange. Relational execution plans can be executed very efficiently since the semantics of the query are known to the processing system, which can apply low-level optimizations. On the other hand, PACT data flows are a bit more general, as they execute user-defined functions inside their operators and allow for multiple data sinks.

3.5 Summary

The PACT programming model is a hybrid of a MapReduce system and an analytical parallel relational database system with the goal to unify the best properties of both.

The PACT extends and generalizes MapReduce in several aspects. It features a more generic tuple data model, a richer set of parallelization primitives, and allows for free composition of operators as acyclic data flows. Consequently, it significantly eases the definition of advanced data analysis programs. By extending MapReduce's concept of parallelizable, system-provided, second-order functions and first-order, user-defined functions, PACT is a superset of MapReduce and applicable to a wide set of data analysis tasks.

Higher-level languages such as Pig, Hive, Jaql, and Cascading, offer similar programming abstractions as PACT but are compiled into workflows of MapReduce programs with limited potential for optimization. In contrast, PACT programs are compiled into parallel data flows. Due to their declarative characteristics, PACT's second-order system functions can be compiled and executed in different ways, which opens potential for optimization. A cost-based optimizer that builds on concepts from relational query optimization chooses the most efficient execution strategies for a PACT program. The ability for optimization is a crucial feature data processing systems. It does not only improve the performance of programs, but also relieves users of making explicit choices for processing algorithm and ensures program independence from data and execution environment. Finally, the compilation of PACT programs to flexible parallel data flows allows us to execute those using efficient processing techniques which are known from relational database systems.

4 Optimizing Data Flows with UDFs

Contents

The PACT programming model (see Chapter 3) is based on operators that consist of a system-provided, second-order function and a user-defined, first-order function. Prior to execution, a cost-based optimizer chooses physical execution strategies for the second-order functions. This step is similar to physical optimization in relational database optimizers.

It is well known from relational database systems that not only the choice of physical operators, but also the order of operators can have a significant impact on a plan's execution time. Optimizers of relational DBMSs reorder or replace relational operators based on algebraic transformation rules during logical optimization. However, applying such plan rewrite transformations on PACT programs is not possible, due to the unknown operator semantics induced by the user-defined functions.

This problem is not unique to the PACT programming model. In fact, many systems such as Hyracks [36], Dryad [133], Spark [207], adopt a data flow abstraction, where a data analysis program is specified as a directed, acyclic graph (DAG) of smaller components that contain arbitrary user code. Even though some of these systems offer higher-level language interfaces [25, 28, 47, 169, 199, 206], support for parallel user-defined functions is a necessity for today's data analysis systems. Even commercial parallel DBMSs, such as Aster Data and Greenplum, have adopted MapReduce-style UDFs [87, 103] to address a broader scope of applications.

The common challenge faced by these systems is to efficiently execute parallel data flows that embed UDFs. This entails parallelization, as well as reordering of operators. These two problems are highly coupled, as the optimal parallelization strategy depends on the operator order and vice versa. Traditional RDBMS optimizers support only UDFs that follow strict templates, such as scalar, aggregation, and table-generator UDFs. Due to these strict templates, the main challenge for RDBMS optimizers is not whether UDFs can be reordered but rather when it is beneficial. In contrast, MapReduce-style UDFs implement much less restrictive templates and hide their semantics inside general-purpose imperative code, a fact that poses new challenges for optimization.

It is well known that query optimization is possible at an abstraction layer where the semantics and the algebraic properties of operators are known. In this chapter, we propose a query optimizer that does not require this assumption. Instead, our optimizer performs a fully automatic static code analysis pass over the UDFs, discovering a handful of properties that guarantee safe reorderings. We observe that a few properties, rather than knowledge of full semantics, are enough to enable many optimizations, including selection and join reordering, as well as limited forms of aggregation push-down. We implement the above concepts in the Stratosphere system [23], and conduct an extensive experimental study. Our experimental results show that we can reproduce most reorderings performed by traditional query optimizers in relational queries, such as join and selection reordering and some forms of aggregation push-down. Further, our system can automatically find optimal plans for non-relational tasks without being informed a priori about the semantics of the operators. While we present our optimizer in the context of the Stratosphere system, the results presented in this chapter are applicable to a variety of parallel data flow systems that use imperative UDFs. This chapter is based our contributions, which were published in References [126, 128, 129].

The remainder of this chapter is organized as follows. In Section 4.1 we present an overview of our approach by exercising operator reordering on a simple example data flow. Section 4.2 discusses conditions to safely reorder MapReduce-style UDFs in data flow programs. In order to evaluate these reorder conditions, a few UDF properties are required. Section 4.3 proposes a technique based on static code analysis (SCA) to automatically extract these properties from UDF code. We present a novel algorithm to enumerate reordered candidate data flows in Section 4.4 and evaluate our approach in Section 4.5. Section 4.6 discusses related work and Section 4.7 summarizes this chapter.

4.1 Solution Overview

Before delving into the details of our solution, we demonstrate the salient points of our complete approach with an example. Assume a PACT program P that consists of three Map operators with first-order functions f_1, f_2, and f_3 interconnected as follows:

$$P : I \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3 \rightarrow O$$

The input data set I contains two integer attributes $\langle A, B \rangle$. The first function f_1 replaces B with $|B|$. The second function f_2 emits all records for which $A \geq 0$ and filters the rest of the records, and the third function f_3 replaces A with the sum $A + B$. For example, with input record $i = \langle 2, -3 \rangle$, the data flow is

$$\langle 2, -3 \rangle \rightarrow f_1 \rightarrow \langle 2, 3 \rangle \rightarrow f_2 \rightarrow \langle 2, 3 \rangle \rightarrow f_3 \rightarrow \langle 5, 3 \rangle$$

while with input record $i' = \langle -2, -3 \rangle$ the data flow is

$$\langle -2, -3 \rangle \rightarrow f_1 \rightarrow \langle -2, 3 \rangle \rightarrow f_2 \rightarrow \perp \rightarrow f_3 \rightarrow \perp$$

where $\perp$ represents the empty list.

Consider now the alternative plan P' where the order of $\texttt{Map}_2$ and $\texttt{Map}_1$ is inverse:

$$P' : I \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_3 \rightarrow O$$

The data flow for records i and i' is

$$\begin{aligned} \langle 2, -3 \rangle &\rightarrow f_2 \rightarrow \langle 2, -3 \rangle \rightarrow f_1 \rightarrow \langle 2, 3 \rangle \rightarrow f_3 \rightarrow \langle 5, 3 \rangle \\ \langle -2, -3 \rangle &\rightarrow f_2 \rightarrow \perp \rightarrow f_1 \rightarrow \perp \rightarrow f_3 \rightarrow \perp \end{aligned}$$

Observe that the order of $\texttt{Map}_1$ and $\texttt{Map}_2$ does not influence the output data set O. Therefore, for input $I = [i, i']$, these two operators can be safely reordered. In fact, if f_2 filters a significant portion of the records in I, this reordering is desirable. On the other hand, f_1 and f_3 cannot be further reordered without changing the result:

$$\langle 2, -3 \rangle \rightarrow f_2 \rightarrow \langle 2, -3 \rangle \rightarrow f_3 \rightarrow \langle -1, -3 \rangle \rightarrow f_1 \rightarrow \langle -1, 3 \rangle$$

We generalize this concept in a safe manner without knowing the semantics of the operators. Our key insight is that *reasoning about the "conflicts" in the data flow suffices to establish reordering conditions*. For example, we do not need to know whether f_3 computes $A+B$ or $A \cdot B$. We only need to know that f_3 replaces the first field of its input record with a new value, which conflicts with f_2 using the first field of its input record to potentially filter some records. We can therefore establish that these operators "conflict" on A, and cannot be reordered. This holds only if *the execution path of a UDF is uniquely determined by its input data*, i. e., communication between two function calls of the same or different UDFs except via the explicitly defined data channels of the data flow program (e. g., shared memory or other forms of communication) is prohibited. We assume this restriction throughout this chapter.

We define a *read set* R_f, and a *write set* W_f for each operator with respect to its UDF f. These sets are defined over *attributes* that need to be extracted from the data flow. In our example flow, we have two attributes A, B, that form the so-called *global record* $\mathsf{A} = \{A, B\}$. The global record provides unique names for all input and intermediate attributes of the data flow. It is explained in more detail in Section 4.2. The read set of an operator contains all attributes that *might influence the operator's output* (except for being merely copied from input to output). The write set of an operator contains all attributes *whose values change with an application of the operator*. We formalize these concepts in Section 4.2. Two operators "conflict" on an attribute if the attribute is contained in both operators' write sets, or in one operator's read set and the other's write set. For example, operator f_1 has $\mathsf{R}_{f_1} = \{B\}$, and $\mathsf{W}_{f_1} = \{B\}$, and operator f_2 has $\mathsf{R}_{f_2} = \{A\}$, and $\mathsf{W}_{f_2} = \emptyset$. These operators do not conflict, and can therefore be reordered.

The next challenge we address is how to derive read and write sets among other necessary properties. In Section 4.3 we present an algorithm that *estimates* these properties using a static code analysis (SCA) pass over the code of the first-order functions. Our technique assumes a fixed API to create records, to read and write record attributes, and to emit records from a UDF. We describe PACT's record API in Section 4.3. Assume the code of the three example first-order functions shown below in the form of 3-address code [6] where the UDFs access fields A and B by their positions (0 and 1 respectively) in the input record:

```
10: f1(InputRecord $ir)
11: $b:=getField($ir,1)
12: $or:=copy($ir)
13: if ($b>=0) goto 16
14: $b:=-$b
15: setField($or,1,$b)
16: emit($or)
17: return
```

(a) UDF f_1

```
20: f2(InputRecord $ir)
21: $a:=getField($ir,0)
22: if($a<0) goto 25
23: $or:=copy($ir)
24: emit($or)
25: return
```

(b) UDF f_2

```
30: f3(InputRecord $ir)
31: $a:=getField($ir,0)
32: $b:=getField($ir,1)
33: $sum:=$a+$b
34: $or=copy($ir)
35: setField($or,0,$sum)
36: emit($or)
37: return
```

(c) UDF f_3

Figure 4.1: 3-Address Code of Example UDFs

The 3-address code of functions f_1, f_2, and f_3 is shown in Figure 4.1. Consider for example the code of function f_2. Recall that f_2 filters records with negative values for attribute A. We can automatically detect that $A \in \mathsf{R}_{f_1}$ by collecting all `getField` statements (in this case instruction `21`), and determine whether the temporary variables introduced (in this case `$a`) are used in the function's code. In our example, instruction `22` uses the value of `$a` in a condition, so we conclude that field 0 of the input record is part of the read set. In the same way, we can detect that $A \in \mathsf{W}_{f_3}$ by looking at instruction `35`, which potentially changes the value of field 0. We can thus conclude that f_2 and f_3 conflict on field 0, and cannot be reordered. This estimation is conservative, but safe. It results in a set of reorderings that all produce the same query result, but it might miss valid reorderings. For example, assume that the input data set I contains only values with $A \geq 0$. Then, instructions `22` and `23` of function f_2 will never be executed, and in fact, f_2 and f_3 can be reordered. However, this is something that cannot be detected by static code analysis, and this reordering will be prohibited by our system.

4.2 Reorder Conditions for MapReduce-style UDFs

In this section we present sufficient conditions to reorder the five operators of the PACT programming model and proof them. PACT's data model and operators are defined in Section 3.1. We start by giving the necessary definitions for our reordering conditions and continue to prove the conditions step-by-step for different pairs of operator types.

4.2.1 Definitions

The user code of an operator accesses record attributes by static field indices. For example line 11 of function f_1 (Figure 4.1(a)) in the previous chapter, reads the attribute with index 1 from the function's input record. Two different functions that both use the same index do not necessarily access the same logical attribute. Instead, an index accesses an attribute with respect to the expected schema of the function's input records. When changing the order of two operators, the schemas of their input records may change such that a field index accesses a different attribute. Therefore, it is essential to avoid that attributes are accessed by wrong indices in order to preserve the original semantics of the data flow. In order to keep track of attributes across user-defined functions, we define the *global record* as a collection of every attribute that is accessed by any operator in the execution plan. It is built by traversing the original data flow from the sources to the sinks. All initial source attributes are added to the global record with a unique name. Each attribute that an operator emits on a field index that was not used in its input is added with a unique name to the global record as well. Thus, the global record includes every attribute of the input data sets as well as the attributes that are created by operators at some stage of the execution plan. Note, the global record is only internally used by the optimizer and not exposed to the user.

Definition 1. The global record A is a unique naming of all base and intermediate attributes in the data flow. In addition, we define I_f and O_f as the set of attributes of the global record A that appear in the input and output of a function f, respectively and $\mathit{Instances}(\mathsf{I}_f)$ as the set of all possible input records of a function f.

Next, we formally define the read and write sets. The write set W_f of a first-order function f contains all attributes whose value might change after applying f.

Definition 2. An attribute A belongs to the write set W_f of a UDF f iff:

$$(1)\ A \notin \mathsf{I}_f \ \wedge\ A \in \mathsf{O}_f$$
$$(2)\ \exists i \in \mathit{Instances}(\mathsf{I}_f) : \pi_A i \neq \pi_A f(i)$$

where π is defined as relational projection with duplicate elimination. The definition captures the fact that an attribute A is in W_f if A is either newly created by f (case 1 of the definition), or that there exists at least one input record i for which f modifies the value of A (case 2 of the definition). The above definition can be extended for UDFs that operate on multiple records.

The read set R_f of a user-defined function f contains all input attributes of f that might influence f's output apart from being merely copied from input to output.

Definition 3. An attribute A belongs to the read set R_f of a UDF f iff:

$$\exists i_1, i_2 \in \mathit{Instances}(\mathsf{I}_f) : \pi_A i_1 \neq \pi_A i_2 \ \wedge \ \pi_{(\mathsf{I}_f \setminus A)} i_1 = \pi_{(\mathsf{I}_f \setminus A)} i_2$$
$$(1)\ |f(i_1)| \neq |f(i_2)|, \text{or}$$
$$(2)\ \pi_{(\mathsf{O}_f \setminus A)} f(i_1) \neq \pi_{(\mathsf{O}_f \setminus A)} f(i_2)$$

where π is defined as relational projection with duplicate elimination. The definition captures the fact that an attribute A can influence f's output if there are two input records i_1 and i_2 that only differ on attribute A for which function f produces different output. It is worth mentioning that the above definitions do not use the semantics of the functions. Section 4.3 discusses how to approximate these sets using static code analysis of the UDFs.

Finally, we define two conditions that are sufficient for reordering of operators in most cases:

Definition 4. Two operators with UDFs f_1, f_2 and key attributes K_1 and K_2 satisfy the *read-only conflict (ROC) condition* iff $(\mathsf{R}_{f_1} \cup \mathsf{K}_1) \cap \mathsf{W}_{f_2} = \mathsf{W}_{f_1} \cap (\mathsf{R}_{f_2} \cup \mathsf{K}_2) = \mathsf{W}_{f_1} \cap \mathsf{W}_{f_2} = \emptyset$.

The ROC condition captures the fact that a UDF does not update or use attributes that another UDF updates. The ROC condition is relevant for all reordering conditions described in this chapter. To reorder KAT operators, we additionally need the condition that key groups are preserved:

Definition 5. A RAT operator with UDF f satisfies the *key group preservation (KGP) condition* for an attribute set $\mathsf{K} \subseteq \mathsf{A}$ iff:

$$\forall i \in \mathit{Instances}(\mathsf{I}_f)$$
$$(1)\ |f(i)| = 1, \text{or}$$
$$(2)\ |f(i)| \leq 1 \ \wedge \ \exists \mathsf{F} \subseteq \mathsf{K}, \forall s_1, s_2 \in \mathit{Instances}(\mathsf{I}_f) : \pi_{\mathsf{F}}(s_1) = \pi_{\mathsf{F}}(s_2) \Rightarrow |f(s_1)| = |f(s_2)|$$

where π is defined as relational projection with duplicate elimination. The KGP condition states that function f, when applied to a set of records I_k with the same values for all attributes in K, either emits or filters all these records, i. e., the filter condition depends on the attributes in K only. The above definition can be extended for KAT operators.

4.2.2 Reordering MapReduce Programs

Reordering Map Operators In Section 4.1 we outlined why two Map operators that satisfy the ROC condition can be reordered without changing the result of the data flow. We now prove this statement formally.

Theorem 1. Two plans, P and P'

$$P : I \rightarrow \texttt{Map}_f \rightarrow S \rightarrow \texttt{Map}_g \rightarrow O$$
$$P' : I \rightarrow \texttt{Map}_g \rightarrow S' \rightarrow \texttt{Map}_f \rightarrow O'$$

consisting of two Map operators $\texttt{Map}_f$ and $\texttt{Map}_g$ with UDFs f and g produce the same result ($O \equiv O'$) if the Map operators satisfy the ROC condition.

Proof. We prove that $O \equiv O'$. Assume a record $i \in I$, and let $O_i = \texttt{Map}_g(\texttt{Map}_f([i]))$, $O'_i = \texttt{Map}_f(\texttt{Map}_g([i]))$, $S_i = \texttt{Map}_f([i]) = f(i)$, and $S'_i = \texttt{Map}_g([i]) = g(i)$. It suffices to prove $\forall i \in I : O_i \equiv O'_i$. We first observe that if the ROC condition holds, the global record can be partitioned as $\mathsf{A} = \overline{\mathsf{W}} \cup (\mathsf{W}_f \dot{\cup} \mathsf{W}_g)$, where $A \dot{\cup} B$ additionally implies that $A \cap B = \emptyset$. We define $\pi_{\mathsf{F}}(r)$ as the projection of record r to attribute subset F.

First, we prove that an invocation of f and g produces the same result cardinality in both plans: $|f(i)| = |f(s'_j)| = k$ where $s'_j \in S'_i$, and $|g(i)| = |g(s_i)| = l$ where $s_i \in S_i$. Records $s'_j \in S'_i$ are produced by applying g to i. Recall that g can only change W_g attributes, therefore $\pi_{\overline{\mathsf{W}} \cup \mathsf{W}_f}(s'_j) = \pi_{\overline{\mathsf{W}} \cup \mathsf{W}_f}(i)$. Observe that the execution path of f depends only on the values of attributes in $\overline{\mathsf{W}} \cup \mathsf{W}_f$. Therefore, f follows the same execution path for s'_j and i, and the cardinality of its output is the same: $\forall s'_j : |f(i)| = |f(s'_j)| = k$. We can similarly prove $|g(i)| = |g(s_i)| = l$. This allows us to decompose plan P for input i as

$$P_1 : i \rightarrow f \rightarrow [s_1, \ldots, s_k]$$
$$P_2 : s_i \rightarrow g \rightarrow [o_{i1}, \ldots, o_{il}] \; \forall i = 1, \ldots, k$$

and plan P' as

$$P'_1 : i \rightarrow g \rightarrow [s'_1, \ldots, s'_l]$$
$$P'_2 : s'_j \rightarrow f \rightarrow [o'_{j1}, \ldots, o'_{jk}] \; \forall j = 1, \ldots, l$$

We will now prove that $\forall i = 1,\ldots,k,\ \forall j = 1,\ldots,l : o_{ij} = o'_{ji}$. We observe that $\pi_{\overline{\mathsf{W}}}(o_{ij}) = \pi_{\overline{\mathsf{W}}}(o'_{ji})$ since attributes in $\overline{\mathsf{W}}$ are not changed by either f or g. Therefore, it suffices to prove that (1) $\pi_{\mathsf{W}_f}(o_{ij}) = \pi_{\mathsf{W}_f}(o'_{ji})$, and (2) $\pi_{\mathsf{W}_g}(o_{ij}) = \pi_{\mathsf{W}_g}(o'_{ji})$. The proofs for the two cases are completely symmetric. We proceed to prove case (1).

From sub-plan P_2 we observe that records o_{ij} are produced by applying g to records s_i. Therefore, they have the same values for all attributes that g does not change: $\pi_{\overline{\mathsf{W}}\cup\mathsf{W}_f}(o_{ij}) = \pi_{\overline{\mathsf{W}}\cup\mathsf{W}_f}(s_i)$. It suffices thus to prove $\pi_{\mathsf{W}_f}(o'_{ji}) = \pi_{\mathsf{W}_f}(s_i)$. Consider sub-plans P_1 and P'_2 that show the application of f to i and s'_j respectively. First, observe that s'_j comes from applying g to i, therefore $\pi_{\mathsf{W}_f}(s'_j) = \pi_{\mathsf{W}_f}(i)$. The execution path of f depends only on values of attributes in $\overline{\mathsf{W}}\cup\mathsf{W}_f$. Since $\pi_{\mathsf{W}_f}(s'_j) = \pi_{\mathsf{W}_f}(i)$, the execution of f in sub-plans P_1 and P'_2 will follow the same execution path. Therefore, the changes applied to i will be the same as the changes applied to s'_j. Therefore, $\pi_{\mathsf{W}_f}(o'_{ji}) = \pi_{\mathsf{W}_f}(s_i)$. □

Reordering Map and Reduce Operators As a next step, we identify a sufficient condition to reorder Map and Reduce operators and proof it. Recall that unlike the MapReduce model, the PACT model allows arbitrary data flows containing Map and Reduce (among other) operators. Assume the plan

$$P : I \rightarrow \texttt{Map}_f \rightarrow S \rightarrow \texttt{Reduce}_g \rightarrow O$$

with input I having two attributes $\langle A,B\rangle$. UDF f emits an input record if both attributes A and B have odd values. UDF g calculates the sum of B using A as key, and appends the sum as a new attribute C to all of its input records. Note that the ROC condition holds. Consider the input data set in the following example application of the plan:

$$\begin{bmatrix}\langle 1,1\rangle, \langle 1,2\rangle,\\ \langle 2,1\rangle, \langle 2,2\rangle\end{bmatrix} \rightarrow \texttt{Map}_f \rightarrow [\langle 1,1\rangle] \rightarrow \texttt{Reduce}_g \rightarrow [\langle 1,1,1\rangle]$$

and the execution if the operators are reordered

$$\begin{bmatrix}\langle 1,1\rangle, \langle 1,2\rangle,\\ \langle 2,1\rangle, \langle 2,2\rangle\end{bmatrix} \rightarrow \texttt{Reduce}_g \rightarrow \begin{bmatrix}\langle 1,1,3\rangle,\\ \langle 1,2,3\rangle,\\ \langle 2,1,3\rangle,\\ \langle 2,2,3\rangle\end{bmatrix} \rightarrow \texttt{Map}_f \rightarrow [\langle 1,1,3\rangle]$$

The ROC condition alone cannot guarantee the reordering of a Map and a Reduce operator. The reason is that the key groups of the Reduce operator in the two plans do not have the same cardinality, and thus result in a different value for attribute C. This would not be a problem if

the Map operator either eliminated whole key groups, or left them intact. Note that if Map also emitted multiple records per call, the cardinality of the key groups would change. Therefore, we need the KGP condition to hold as well.

Theorem 2. Two plans P and P'

$$P : I \rightarrow \texttt{Map}_f \rightarrow S \rightarrow \texttt{Reduce}_g \rightarrow O$$
$$P' : I \rightarrow \texttt{Reduce}_g \rightarrow S' \rightarrow \texttt{Map}_f \rightarrow O'$$

consisting of a Map operator $\texttt{Map}_f$ and a Reduce operator $\texttt{Reduce}_g$ with UDFs f and g produce the same result ($O \equiv O'$) if both operators satisfy the ROC condition and the KGP condition holds for holds for f and the key K of the Reduce operator.

Proof. We prove that $O \equiv O'$. Let $I = \cup_{\mathsf{k}} I_{\mathsf{k}}$, where I_{k} is the key group with key value k, and the plans

$$P : I_{\mathsf{k}} \rightarrow \texttt{Map}_f \rightarrow S_{\mathsf{k}} \rightarrow \texttt{Reduce}_g \rightarrow O_{\mathsf{k}}$$
$$P' : I_{\mathsf{k}} \rightarrow \texttt{Reduce}_g \rightarrow S'_{\mathsf{k}} \rightarrow \texttt{Map}_f \rightarrow O'_{\mathsf{k}}$$

It suffices to prove that $O_{\mathsf{k}} \equiv O'_{\mathsf{k}}$. Observe that if the KGP condition holds for f and K, $|S_{\mathsf{k}}| = |I_{\mathsf{k}}|$, or $|S_{\mathsf{k}}| = 0$. If $|S_{\mathsf{k}}| = 0$, then $\texttt{Map}_f$ will also filter all records from S'_k in P', and trivially $O_{\mathsf{k}} \equiv O'_{\mathsf{k}} = \perp$. Assume that $|I_{\mathsf{k}}| = |S_{\mathsf{k}}| = k$, and $|O_{\mathsf{k}}| = l$. Since the Reduce UDF treats I_{k} in P' in the same way as S_{k} in P (because $|I_k| = |S_k|$ and the ROC condition holds), and the Map UDF emits exactly one record per input, it holds that $|S'_{\mathsf{k}}| = |O'_{\mathsf{k}}| = l$. Therefore, we can decompose plan P as

$$P_1 : \quad \forall i \in [i_1, \ldots, i_k],\ i \rightarrow f \rightarrow s,\ s \in [s_1, \ldots, s_k]$$
$$P_2 : \quad [s_1, \ldots, s_k] \rightarrow g \rightarrow [o_1, \ldots, o_l]$$

and plan P' as

$$P'_1 : \quad [i_1, \ldots, i_k] \rightarrow g \rightarrow [s'_1, \ldots, s'_l]$$
$$P'_2 : \forall s' \in [s'_1, \ldots, s'_l],\ s' \rightarrow f \rightarrow o',\ o' \in [o'_1, \ldots, o'_l]$$

We now prove that $\forall j, j = 1,\ldots,l : o_j = o'_j$. Due to the ROC condition it suffices to prove (1) $\pi_{\mathsf{W}_f}(o_j) = \pi_{\mathsf{W}_f}(o'_j)$, and (2) $\pi_{\mathsf{W}_g}(o_j) = \pi_{\mathsf{W}_g}(o'_j)$.

We proceed to prove case (1). Case (2) is proven similarly. From sub-plan P_2, and record o_j, there is a record s_x with the same attribute values for W_f: $\forall j, j = 1,\ldots,l\ \exists x, x = 1,\ldots,k :$ $\pi_{\mathsf{W}_f}(o_j) = \pi_{\mathsf{W}_f}(s_x)$. Note that Reduce may "consolidate" multiple records into one, or produce multiple records per input record. However, due to the ROC condition, attributes in W_f must be preserved. Similarly, from P'_1 we have $\forall j, j = 1,\ldots,l\ \exists y, y = 1,\ldots,k : \pi_{\mathsf{R}_f}(s'_j) = \pi_{\mathsf{R}_f}(i_y)$ (due to the ROC condition, attributes in R_f are preserved as well). Using the same arguments as in the proof of Theorem 1, we know that g follows the same execution path in sub-plans P_2 and P'_1.

Therefore, f follows the same execution path for records s'_j and i_x, so the result records of applying f to these records will also share the same values for W_f attributes: $\pi_{\mathsf{W}_f}(o'_j) = \pi_{\mathsf{W}_f}(s_x) \Rightarrow$ $\pi_{\mathsf{W}_f}(o'_j) = \pi_{\mathsf{W}_f}(o_j)$. □

The condition for reordering two Reduce operators are the ROC condition and the KGP condition for both UDF-key pairs. The proof proceeds similarly.

4.2.3 Reordering Binary Second-Order Functions

After having presented sufficient conditions to reorder Map and Reduce operators, we continue to include the binary operators of the PACT programming model, namely Cross, Match, and CoGroup. We start to give conditions for RAT operators and look at KAT operators subsequently.

Record-at-a-time Operators We first cover plans with RAT operators that are constructed using the Cross, Match, and Map PACTs. Assume a Cross operator with UDF f and inputs R, S. The operator applies f to every pair $(r,s) \in R \times S$. Here, the Cartesian product $R \times S$ of two data sets $R = [r_1,\ldots,r_n]$, $S = [s_1,\ldots,s_m]$ is defined as a data set $R \times S = [r_i|s_j : i = 1\ldots,n, j = 1,\ldots,m]$ where $r|s$ is the concatenation of records r and s. The attribute set of $R \times S$ is the union of the attribute sets of R and S with a proper renaming (e. g., each attribute is prefixed by the data set name it belongs to).

We observe that we can *conceptually* transform a Cross operator to a Map operator with the same UDF over the Cartesian product:

$$\texttt{Cross}_f(R,S) \equiv \texttt{Map}_f(R \times S)$$

We can similarly transform a Match operator with UDF f to a Map operator with UDF f' over the Cartesian product:

$$\texttt{Match}_f(R,S) \equiv \texttt{Map}_{f'}(R \times S)$$

The difference here is that we need to change the UDF f in order to incorporate the implicit equi-join performed by the Match second-order function. Assume that the join keys are attributes $R.A, S.B$. We substitute f with

$$f'(r|s) = \texttt{if } (R.A = S.B) \texttt{ then } f(r,s) \texttt{ else } \bot$$

We stress that this is a conceptual transformation in order to establish reordering conditions; all optimizations described in this chapter are non-intrusive. This transformation simply means that the attributes used as keys for the Match operator are added to the read set R_f of UDF f to construct the read set of the function $\mathsf{R}_{f'}$.

Using the above transformations, plans that contain Match, Cross, and Map operators are equivalent to plans that contain only Map operators and Cartesian products. Therefore, it only remains to establish when the latter two can be reordered:

Theorem 3. A Map operator with UDF f and a Cartesian product operator $R \times S$ can be reordered as

$$\texttt{Map}_f(R \times S) \equiv \texttt{Map}_f(R) \times S$$

iff $(\mathsf{R}_f \cup \mathsf{W}_f) \cap \mathsf{S} = \emptyset$, where S is the attribute set of S. The case of pushing the operator to the other side of the Cartesian product is symmetric.

The proof follows directly from the fact that $(\mathsf{R}_f \cup \mathsf{W}_f) \cap \mathsf{S} = \emptyset \Rightarrow f(r|s) = f(r)|s$.

It is straightforward to construct the conditions that allow reordering for Match, Cross, and Map operators using Theorems 1 and 3. We now show the proof for reordering two Match operators with first order functions f, g, and key attributes $\mathsf{K}_f, \mathsf{K}_g$ as a series of transformations as shown in Figure 4.2

Step (a) $\rightarrow$ (b) substitutes the Match operators with their Map and Cartesian product equivalents. Step (b) $\rightarrow$ (c) reorders $\texttt{Map}_{f'}$ with the Cartesian product with T. For plans (b) and (c) to be equivalent it is necessary that f' does not use attributes of T ($(\mathsf{R}_{f'} \cup \mathsf{W}_{f'}) \cap \mathsf{T} = \emptyset$). Step (c) $\rightarrow$ (d) makes use of the conditions of Theorem 1 (namely the ROC condition on UDFs f', g') to reorder the two Map operators, and reorders the two Cartesian products using the normal associativity rule. Step (d) $\rightarrow$ (e) pushes $\texttt{Map}_{g'}$ under the Cartesian product, requiring the condition $(\mathsf{R}_{g'} \cup \mathsf{W}_{g'}) \cap \mathsf{R} = \emptyset$. Finally, step (e) $\rightarrow$ (f) reconstructs the Match operators. By collecting the conditions needed by the series of transformations, we arrive at the conditions to reorder two Match operators.

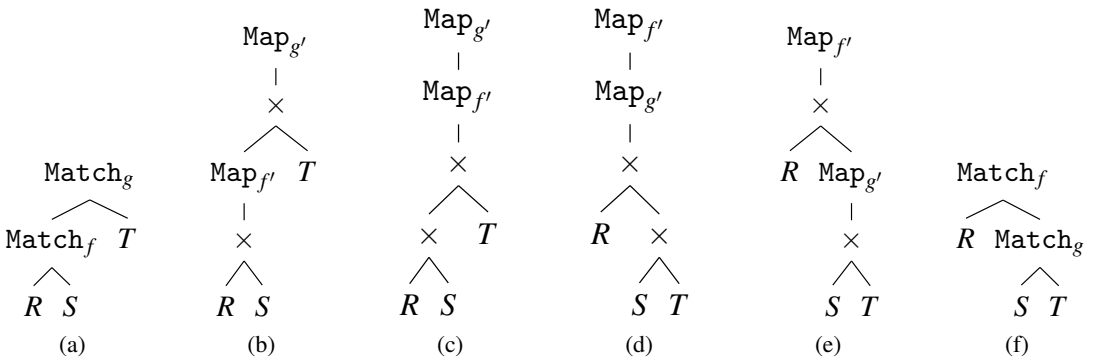


Figure 4.2: Reorder transformation of two Match operators. Requires that f' and g' fulfill the ROC condition and $(\mathsf{R}_{g'} \cup \mathsf{W}_{g'}) \cap \mathsf{R} = \emptyset$.

Lemma 1. Two Match operators with UDFs f, g and key sets $\mathsf{K}_f \subset \mathsf{R} \cup \mathsf{S}, \mathsf{K}_g \subset \mathsf{S} \cup \mathsf{T}$ can be reordered iff the ROC condition holds for f', g', $(\mathsf{R}_{f'} \cup \mathsf{W}_f) \cap \mathsf{T} = \emptyset$, and $(\mathsf{R}_{g'} \cup \mathsf{W}_g) \cap \mathsf{R} = \emptyset$ where $\mathsf{R}_{f'} = \mathsf{R}_f \cup \mathsf{K}_f$, and $\mathsf{R}_{g'} = \mathsf{R}_g \cup \mathsf{K}_g$.

By repeating the same process for each pair of Match, Cross, and Map, we establish similar conditions for all combinations of these operators.

Key-at-a-time Operators Incorporating KAT operators (Reduce and CoGroup) requires stricter conditions, since groups must be preserved. We first show how to reorder a Reduce operator with a Cartesian product.

Theorem 4. A Reduce operator with UDF g and key $\mathsf{K} \cup \mathsf{S}$ and a Cartesian product operator $S \times T$ can be reordered as

$$\texttt{Reduce}_{g,\mathsf{K}\cup\mathsf{S}}(S \times T) \equiv S \times \texttt{Reduce}_{g,\mathsf{K}}(T)$$

iff $(\mathsf{R}_g \cup \mathsf{W}_g) \cap \mathsf{S} = \emptyset$ and all records of S are unique, i. e., $S = [s_1, \ldots, s_n]$, with $i, j = 1, \ldots, n, i \neq j \Rightarrow s_i \neq s_j$.

Proof. Assume the data sets $S = [s_1, \ldots, s_n]$, with $i, j = 1, \ldots, n, i \neq j \Rightarrow s_i \neq s_j$ and $T = [t_1, \ldots, t_m]$. The key of the Reduce operator $\mathsf{K} \cup \mathsf{S}$ includes all attributes of data set S. Note that $\mathsf{K} \subset \mathsf{T}$. Every record of the Cartesian product can be written as $s_i|k_j|t'_k$, where k_j is the part of the T record with attributes K, and t'_k is the part of an T record with non-key attributes. Every record $s_i|k_j|t'_k$ of the Cartesian product belongs to the same Reduce group G_{ij}, determined by s_i and k_j only. The output of the plan is $[g(G_{ij}), G_{ij} = \{s_i|k_j|t'_k\}]$. Assume that g does not use any attribute of S for any purpose other than grouping its input data set. Then, it is safe to "push" $\texttt{Reduce}_g$ to the data set T and remove the S part of the Reduce key. This will produce groups $G_j = \{k_j|t'_k\}$,

and the output of the Reduce operator will be $[g(G_j)]$. By performing the Cartesian product of these groups with S, we get the set of records $s_i|g(G_j)$. If the Reduce UDF g simply emits the S attributes unchanged, we have $s_i|g(k_j|t_k) = g(s_i|k_j|t_k)$. □

Using the above transformation, we can, in principle, reorder Reduce with Match and Cross operators by transforming the latter to Map operators over Cartesian products. It is rather seldom that the Reduce key includes all attributes of a data set. However, we can consider special cases where it is safe to *add* the S attributes to the Reduce key without changing the result. One case is when $|S| = 1$. This appears quite often in practice when implementing SQL queries with correlated subqueries that return a single tuple. More interestingly, using Theorem 4 as basis, we can arrive at a Match-Reduce transformation similar to the invariant grouping transformation in relational DBMSs [51]. Assume the plan

$$\text{(a) } \texttt{Reduce}_{g,\mathsf{F}}(\texttt{Match}_{f,S.\mathsf{K}=T.\mathsf{F}}(S,T))$$

where the Match keys are $\mathsf{K} \subset \mathsf{S}$, $\mathsf{F} \subset \mathsf{T}$, and the Reduce key is a superset of F. Assume that each record in S is uniquely identified by its K attributes[1]. Then, in every record received by the Reduce operator, the F part uniquely determines all S attributes due to the join condition implied by the Match keys. We can therefore add S to the key of the Reduce operator without changing the Reduce groups, and apply Theorem 4 to push the Reduce under the Match. As always, the ROC and KGP conditions must hold in order to reorder the Reduce and Map UDFs. The transformation steps taking plan (a) above as the starting point are shown in Figure 4.3.

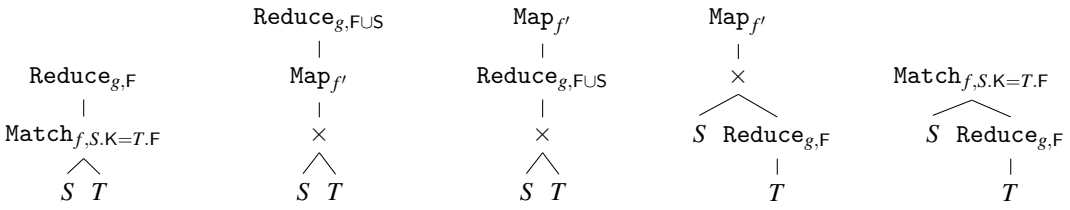


Figure 4.3: Reorder transformation of a Match and a Reduce operator. Requires that (1) the ROC condition holds for f' and g, (2) f' fulfills the KGP condition for $\mathsf{F} \cup \mathsf{S}$, (3) $(\mathsf{R}_g \cup \mathsf{W}_g) \cap \mathsf{S} = \emptyset$, and (4) records in S are uniquely identified by their K attributes.

The last step is to incorporate CoGroup operators. We note that a CoGroup operator can be conceptually transformed to a Reduce operator over the *tagged union* $S \cup^* T$ of its inputs S, T:

$$\texttt{CoGroup}_g(S,T) \equiv \texttt{Reduce}_{g'}(S \cup^* T)$$

[1] Uniqueness properties of attribute sets can be passed to the PACT optimizer via optimizer hints.

The tagged union of two data sets S and T is simply the data set S followed by the data set T, where each record has an additional *lineage* attribute l, which tracks the data set that the record originates from. The CoGroup UDF g is properly annotated to distinguish between data sets based on the lineage attribute, yielding the Reduce UDF g'.

Map and Reduce operators can be pushed down under the tagged union $S \cup^* T$ if their UDFs operate only on one of the tagged union's inputs. This can be properly detected using the lineage attribute l. For example, assume that we want to push a Map operator with UDF f under the tagged union $S \cup^* T$, and that f uses only S attributes. We can define a UDF f_S as

$$f_S(s) = \begin{cases} f(s) \text{ if } s.l = S \\ s \text{ otherwise.} \end{cases}$$

thus forcing the Map UDF f to ignore T records. This transformation yields

$$\texttt{Map}_{f_S}(S \cup^* T) \equiv \texttt{Map}_f(S) \cup^* T$$

and allows the following series of transformations that show how a Map operator can be reordered with a CoGroup operator as depicted in Figure 4.4.

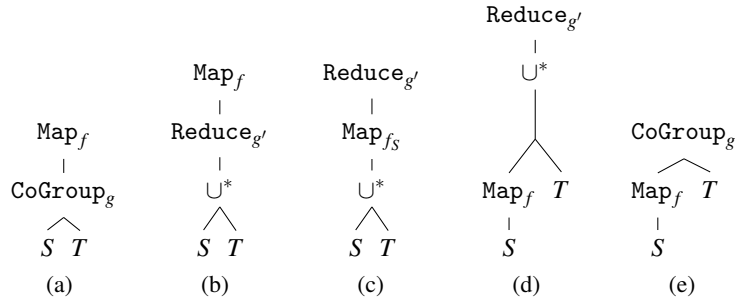


Figure 4.4: Reorder transformation of a Map and a CoGroup operator. Requires that (1) the ROC condition holds for f_S and g', (2) f_S fulfills the KGP condition for the grouping key of the Reduce operator, and (3) $(\mathsf{R}_f \cup \mathsf{W}_f) \cap \mathsf{T} = \emptyset$.

Step (a) $\rightarrow$ (b) replaces CoGroup with its Reduce equivalent. Step (b) $\rightarrow$ (c) uses the conditions of Theorem 2 to reorder the Map and Reduce operators and transforms Map's UDF f to f_S. Step (c) $\rightarrow$ (d) pushes the Map operator under the tagged union by reversing the previous transformation. Finally, step (d) $\rightarrow$ (e) reconstructs the CoGroup operator and transforms f_S back to f.

4.2.4 Possible Optimizations

We have presented sufficient conditions to reorder pairs of PACT operators. These conditions are usually the ROC and the KGP conditions, together with some restrictions on the key of the Reduce operator.

These conditions lead to a number of possible optimizations. First, assuming a straightforward implementation of an acyclic SQL query as a PACT program, our conditions enable many of the join and selection reorderings that RDBMS optimizers perform. Second, we enable the invariant grouping transformation [51], the most elementary form of aggregation push-down. More advanced transformations that include group-by considered by RDBMSs assume knowledge of the nature of the aggregating function, and are thus of limited applicability in settings of arbitrary UDFs as ours [52].

We do not support reorderings that need semantic information to be established, including associative side-effects. For example, we cannot reorder two Map functions that add a constant number to the same field. In addition, the fact that we evaluate the sufficient conditions for reordering through static code analysis poses further restrictions to the possible optimizations (see Section 4.3 for details).

Record API method	Description
`$or=new OutputRecord()`	Creates an empty output record `$or`.
`$or=new OutputRecord($ir)`	Creates a new output record `$or` by copying an input record `$ir`.
`$or2=new OutputRecord($or,$ir)`	Creates a new output record `$or2` by merging an input record `$ir` and an output record `$ir`.
`$t:=getField($ir,n)`	Reads the *n*-th field of input record `$ir` into the temporary variable `$t`.
`setField($or,n,$t)`	Sets the *n*-th field of output record `$or` to the value of the temporary variable `$t`.
`setField($or,n, null)`	Projects the *n*-th field of output record `$or` by setting its value to `null`.
`emit($or)`	Emits the output record `$or` from the user-defined function.

Table 4.1: Record API methods

4.3 Obtaining Reordering Information with Static Code Analysis

The reordering proofs presented in the previous Section 4.2 assume knowledge of a global record, read and write sets for each operator, as well as bounds on the output cardinality. In this section, we present our solution to conservatively estimate this information using static code analysis (SCA) techniques.

Our solution relies on a static code analysis framework to get the bytecode of the analyzed UDF, for example as typed three-address code [6]. The framework must provide a control flow graph (CFG) abstraction, in which each code statement is represented by one node along with a function PREDS(s) that returns the statements in the CFG that are "true" predecessors of statement s, i.e., they are not both predecessors and descendants. Finally, the framework must provide two methods DEF-USE(s,`$v`) and USE-DEF(s,`$v`) that represent the Definition-Use chain of the variable `$v` at statement s, and the Use-Definition chain of variable `$v` at statement s respectively. The Use-Definition chain USE-DEF(s,`$v`) of a statement s and variable `$v` is a list of all possible definitions of variable `$v` that reach s without being overridden by other definitions. The Definition-Use chain DEF-USE(s,`$v`) of a statement s and a variable `$v` is a list of all uses of `$v` as defined by statement s. Any SCA framework that provides these abstractions can be used.

For the remainder of this section, we assume that the UDF code is formatted as typed three-address code [6]. The possible statements in three-address code are definitions of a local (e. g., `int i`) or a temporary (e. g., `int $t`) variable, assignment (e. g., `$t:=3`), branching (e. g.,

Algorithm 1 Read-Set approximation for a UDF f

```
1: function APPROXIMATE-READ-SET(f)
2:     R_f = ∅
3:     G = all statements of the form g:$t=getField($ir,n) in function f
4:     for g in G do
5:         if DEF-USE(g, $t) ≠ ∅ then R_f = R_f ∪ {n}
6:     return R_f
```

`if ($t<3) goto label`), as well as basic arithmetic and method calls. In addition, we assume the existence of an attribute type, `Attribute`, as well as record types `InputRecord`, and `OutputRecord`, and a set of methods that operate on these types, i. e., methods to create, access, modify, and emit records from user-defined functions. These methods constitute the assumed Record API, which is exposed to the programmer of PACT programs. Table 4.1 lists the methods of the Record API.

Since the Record API accesses record fields by index, we need a mapping from field indices to the attributes of the global record A. For this purpose we define an *input redirection map* $\alpha(f,n)$, which maps every field index $n \in \mathbb{N}$ of the input of a function f to the corresponding attribute in the global record A. Equivalently, we define an *output redirection map* $\omega(f,n)$, which maps every field index $n \in \mathbb{N}$ of the output of a function f to the corresponding attribute in the global record A.

We continue to present our solution to approximate read sets, write sets, and bounds on the output cardinality of user-defined functions using static code analysis techniques. Finally, we discuss the safety of our approach to automatically extract information from user code.

4.3.1 Estimating Read Sets

We estimate the read set R_f of an operator by scanning its UDF's code for statements of the form `s:$t:=getField($ir,n)`. We assume that integer `n` is statically computable. Recall that the n-th field of an input record of a function f corresponds to attribute $\alpha(f,n)$ of the global record. We then look up all uses of the temporary variable `$t` in the code using the data structure DEF-USE(`$t`). If such uses exist, then we add the attribute $\alpha(f,n)$ to R_f.

Algorithm 1 gives the pseudocode to extract read set information from a UDF f. The algorithm starts with an empty read set R_f (line 2). Subsequently, all statements of the form `$t := getField($ir, n)` are enumerated (line 3). Each statement refers to a field `n` that is read into a variable `$t`. A field `n` is added to the read set R_f if its corresponding variable `$t` is somewhere used in the UDF (line 5).

4.3.2 Estimating Write Sets

Estimating the write set W_f of an operator is more challenging than read set estimation since also implicit modifications must be taken into account. Our Record API provides two constructors to create an output record `$or`. First, a copy constructor `$or=new OutputRecord($ir)` to copy an input record `$ir`. Second, the default constructor `$or=new OutputRecord()` to create a new and empty output record `$or`. The subtle difference is that the first constructor implicitly copies all attributes of the input record (*Implicit Copy*) while the second method implicitly projects all attributes (*Implicit Projection*). In addition, the API provides methods to explicitly copy, project, modify, and add single attributes to output records. Therefore, the code analysis method to estimate write sets must identify whether a user function implicitly copies or projects, and estimate a complementary set of explicitly projected or copied attributes. In addition, a set of modified and added attributes must be derived.

In order to identify the implicit operation and the attribute sets required for the write set estimation, we start by collecting all statements of the form `e:emit($or)`, which emit the output record `$or`. We track the origin of `$or` and can safely identify the implicit operation by identifying the constructor call. If both constructors are used in different code paths, implicit projection is the safe choice. Subsequently, the remaining attribute sets are estimated by collecting all statements `s:setField($or,n,$t)`. Explicit projections can be identified if `$t` is `null`. Explicit copies require that `$t` was previously set by `l:$t:=getField($ir,n)`. This can be easily detected by looking at USE-DEF(`$t`). In all other cases, statement `s` defines an explicit modification operation and is added to the appropriate set. Note that it is always safe to consider `s` as an explicit modification. Our implementation includes an additional record constructor `$or=new OutputRecord($or,$ir)` that creates a new output record by merging the fields of an input record and an output record in order to support efficient binary UDFs. This constructor yields an implicit copy operation for the input record.

By looking at all statements `s`, we can also keep track of the global record. An attribute $B = \beta(f,n)$ at index n of an output record of a function f is added to the global record if it is not included in the set of input attributes of f, i. e., $B \notin I_f$.

Algorithm 2 shows the pseudocode to approximate, i. e., compute a superset of the write set W_f of a UDF f. It computes four sets of indices, O_f, E_f, C_f, P_f from which eventually an approximation of f's write set W_f is assembled. The *origin set* O_f of UDF f is a set of input indices. An index $o \in O_f$ means that all fields of the o-th input record of f are copied verbatim to the output. This is necessary to track the origin of fields in case of UDFs with more than one input, such as Cross or Match. The *explicit modification set* E_f contains the indices of all fields that are modified and then included in the output. The *copy set* C_f contains the indices of all fields that are copied verbatim from one input record to the output. Finally, the *projection set* P_f contains fields that are projected from the output, by explicitly being set to `null`. The write set is computed from these sets using the function ASSEMBLE-WRITE-SET (lines 35-39). All fields in E_f

Algorithm 2 Write-Set approximation for a UDF f

function APPROXIMATE-WRITE-SET(f)
  E = all statements of the form `e:emit($or)` in function f
  (O_f, E_f, C_f, P_f) = VISIT-STMT(ANY(E), `$or`)
  for `e` in E **do**
    (O_e, E_e, C_e, P_e) = VISIT-STMT(`e`, `$or`)
    (O_f, E_f, C_f, P_f) = MERGE($(O_f, E_f, C_f, P_f), (O_e, E_e, C_e, P_e)$)
  return ASSEMBLE-WRITE-SET(f, O_f, E_f, C_f, P_f)
function VISIT-STMT(`s`, `$or`)
  if VISITED(`s`, `$or`) **then**
    return MEMO-SETS(`s`, `$or`)
  VISITED(`s`, `$or`) = **true**
  if `s` of the form `$or = new OutputRecord()` **then**
    return $(\emptyset, \emptyset, \emptyset, \emptyset)$
  if `s` of the form `$or = new OutputRecord($ir)` **then**
    return (INPUT-ID(`$ir`), $\emptyset, \emptyset, \emptyset$)
  P_s = PREDS(`s`)
  (O_s, E_s, C_s, P_s) = VISIT-STMT(ANY(P_s), `$or`)
  for `p` in P_s **do**
    (O_p, E_p, C_p, P_p) = VISIT-STMT(`p`, `$or`)
    (O_s, E_s, C_s, P_s) = MERGE($(O_s, E_s, C_s, P_s), (O_p, E_p, C_p, P_p)$)
  if `s` of the form `$or = new OutputRecord($or, $ir)` **then**
    return $(O_s \cup$ INPUT-ID(`$ir`)$), E_s, C_s, P_s$)
  if `s` of the form `setField($or, n, $t)` **then**
    T =USE-DEF(`s`, `$t`)
    if all `t` $\in T$ of the form `$t=getField($ir,n)` **then**
      return $(O_s, E_s, C_s \cup \{n\}, P_s)$
    else
      return $(O_s, E_s \cup \{n\}, C_s, P_s)$
  if `s` of the form `setField($or, n, null)` **then**
    return $(O_s, E_s, C_s, P_s \cup \{n\})$
function MERGE($(O_1, E_1, C_1, P_1), (O_2, E_2, C_2, P_2)$)
  $C = (C_1 \cap C_2) \cup \{x | x \in C_1,$ INPUT-ID$(x) \in O_2\}$
    $\cup \{x | x \in C_2,$ INPUT-ID$(x) \in O_1\}$
  return $(O_1 \cap O_2, E_1 \cup E_2, C, P_1 \cup P_2)$
function ASSEMBLE-WRITE-SET(f, O_f, E_f, C_f, P_f)
  $W_f = E_f \cup P_f$
  for $i \in$ INPUTS(f) **do**
    if $i \notin O_f$ **then** $W_f = W_f \cup ($INPUT-FIELDS$(f, i) \setminus C_f)$
  return W_f

and P_f are explicitly modified or set to `null` and therefore in W_f. For inputs that are not in the origin set O_f, we add all fields of that input which are not in C_f, i. e., not explicitly copied.

To derive the four sets, function APPROXIMATE-WRITE-SET finds all statements of the form `e: emit($or)` in function f, which include the output record `$or` in the output (line 2). It then calls for each statement `e` the recursive function VISIT-STMT that recurses from statement `e` backwards in the control flow graph (lines 3-6). The function performs a combination of reverse data flow and control flow analysis but does not change the values computed for statements once they have been determined. The function ANY returns an arbitrary element of a set.

The interesting part of the algorithm is listed between lines 16-30 of the algorithm. First, the algorithm finds all predecessor statements of the current statement, and recursively calls VISIT-STMT. The sets are merged using the MERGE function (lines 31-34). MERGE provides a conservative approximation of these sets, by creating maximal E,P sets, and minimal O,C sets. This guarantees that the data conflicts that will arise are a superset of the true conflicts in the program. When a statement of the form `setField($or, n, null)` is found (line 29), field n of the output record is explicitly projected, and is thus added to the projection set P. When a statement of the form `setField($or, n, $t)` is found (line 23), the USE-DEF chain of `$t` is checked. If the temporary variable `$t` came directly from field n of the input, it is added to the copy set C, otherwise it is added to the explicit write set E. When we encounter a statement of the form `$or = new OutputRecord()` (line 12), we have reached the creation point of the output record, where it is initialized to the empty record. The recursion then ends. Another base case is reaching a statement `$or = new OutputRecord($ir)` (line 14) where the output record is created by copying all fields of the input record `$ir`. This adds the input id of record `$ir` to the origin set O. A `$or = new OutputRecord($or, $ir)` statement (line 21) results in an inclusion of the input id of the input record `$ir` in the origin set O, and a further recursion for the output record `$or`. The algorithm maintains a memo table MEMO-SETS to support early exit of the recursion in the presence of loops (line 9). The memo table is implicitly updated at every `return` statement of VISIT-STMT.

Function VISIT-STMT always terminates in the presence of loops in the UDF code, since it will eventually find the statement that creates the output record, or visit a previously seen statement. This is due to PREDS always exiting a loop after visiting its first statement. Thus, loop bodies are only visited once by the algorithm. The complexity of the algorithm is $O(en)$, where n is the size of the UDF code, and e the number of emit statements. This assumes that the Use-Def and Def-Use chains have been precomputed.

When estimating write sets, we need to distinguish between record-at-a-time and key-at-a-time operators. While RAT operators receive only a single input record per input, KAT operators such as Reduce and CoGroup receive a group of records per input. All records of a group can have different attributes values, except for the key attributes on which the records are grouped. Hence, key attributes are the only attributes which can be possibly constant in KAT operators. When estimating the write set of an operator, we compute a super set of the actual write set. For

KAT operators, the estimated write set always includes all non-key attributes. This guarantees that only key attributes are treated as constant.

4.3.3 Estimating Output Cardinality Bounds

The lower and upper bound on the output cardinality of a UDF can be derived by another pass over the UDF code. We determine the bounds for each emit statement e and combine those to derive the bounds of the UDF. For the lower bound $\lfloor EC_f \rfloor$, we check whether there is a statement before statement e that jumps to a statement after e. If there is none, the emit statement will always be executed and we set $\lfloor EC_f \rfloor = 1$. If such a statement exists, statement e could potentially be skipped during execution, so we set $\lfloor EC_f \rfloor = 0$. For the upper bound $\lceil EC_f \rceil$, we determine whether there is a statement after e that can jump to a statement before e. If yes, the statement could be executed several times during the UDF's execution, so we set $\lceil EC_f \rceil = +\infty$. If such a statement does not exist, statement e can be executed at most once so we set $\lceil EC_f \rceil = 1$. To combine the bounds we choose for the lower bound of the UDF the highest lower bound over all emit statements and for the upper bound the highest upper bound over all emit statements.

Algorithm 3 Computation of output cardinality bounds for a UDF f

```
1:  function APPROXIMATE-OUTCARD-BOUNDS(f)
2:      E = all statements of the form e:emit($or) in function f
3:      ⌊EC_f⌋, ⌈EC_f⌉ = 0
4:      for e in E do
5:          (⌊EC_e⌋, ⌈EC_e⌉) = GET-CALL-BOUNDS(e)
6:          if ⌊EC_f⌋ < ⌊EC_e⌋ then ⌊EC_f⌋ = ⌊EC_e⌋
7:          if ⌈EC_f⌉ < ⌈EC_e⌉ then ⌈EC_f⌉ = ⌈EC_e⌉
8:      return (⌊EC_f⌋, ⌈EC_f⌉)
9:  function GET-CALL-BOUNDS(e)
10:     ⌊EC_e⌋, ⌈EC_e⌉ = 1
11:     C = all statements of the form g:if (condition) goto(l) in function f
12:     for c in C do
13:         j = the statement that the condition statement c jumps to
14:         if (c ∈ PREDS(e)) AND (e ∈ PREDS(j)) then ⌊EC_e⌋ = 0
15:         if (j ∈ PREDS(e)) AND (e ∈ PREDS(c)) then ⌈EC_e⌉ = +∞
16:     return (⌊EC_e⌋, ⌈EC_e⌉)
```

Algorithm 3 gives the pseudocode to extract output cardinality information from a user-defined function. APPROXIMATE-OUTCARD-BOUNDS determines all statements that emit records from the user-defined function, i. e., all statements of the form `emit($or)` (line 2). For each of these statements, the function GET-CALL-BOUNDS returns individual output cardinality bounds (line 5). The bounds of the user-defined function are computed as the maximum of all individual

statements bounds (lines 6 and 7) and returned (line 8). Function GET-CALL-BOUNDS determines the output cardinality bounds of an emit statement by checking for each conditional statement (lines 11-12) whether it might jump behind the emit statement (line 14) or jump in front of it (line 15). The former might result in skipping the statement, i. e., a lower bound of 0, and the latter might cause a repeated execution of the statement, i. e., an upper bound of $+\infty$.

4.3.4 Guaranteeing Safety

The most important property of any method that relies on static code analysis is to guarantee *safety*. In our setting, safety is defined as follows: Our analysis algorithm creates a set of properties, which in turn lead to a certain set of possible reorderings. These reorderings result in a set of plans P' equivalent to the initial plan p. Our method is safe if each $p', p' \in P'$ and p produce the same query result for every possible input I.

We guarantee safety through conservatism. In particular, we guarantee that the properties discovered by our static code analysis algorithm are *supersets* of the true properties of any execution of the program for any collection of input data sets. We achieve this by considering all possible execution paths of operators, and adding an attribute to the global record, and the read and write set of an operator when in doubt. Since the discovered properties are supersets of the real properties, they cause additional conflicts (see Section 4.2) leading to a subset of the valid reorderings, and thus to a subset of the true equivalent alternative plans.

4.4 Plan Enumeration

In this section, we present an algorithm that, for a given data flow, enumerates all data flows that can be derived by valid pairwise reorderings of operators. The algorithm differs significantly from the well-known enumeration algorithms used in traditional relational database optimizers, namely enumeration via top-down, branch-and-bound [82, 97] or bottom-up, dynamic programming [163, 186]. This is due to the difference in the algorithm's input. Traditional relational optimizers operate on algebraic expressions on which heuristics such as selection and projection push-down can be applied and from which data structures such as join graphs can be derived. In contrast, the PACT optimizer does not receive a logical representation of the program but only a specific data flow instance. The enumeration algorithm must be able to generate all valid reordered data flows given this specific data flow instance. In the presented version, the algorithm is restricted to tree-shaped data flows, i. e., an operator may only have a single ancestor.

Algorithm 4 provides pseudocode to enumerate all valid alternatives for a given data flow. The algorithm is based on recursive calls to enumerate alternatives for sub-flows and the exchange of two neighboring operators. In the listing, data flows and sets of data flows are denoted with capitalized names while operators and sets of operators have lowercased names. The functions `getRoot`(D) and `rmRoot`(D) return or remove the root of the data flow D, while `addRoot`(D, r) appends r as root of D and `setRoot`(D, r) replaces D's root with r. For ease of exposition, the algorithm as shown handles data flows with single-input operators only. However, it can be easily extended to deal with non-unary operators, and our implementation can, in fact, handle binary operators.

We discuss the algorithm and argue that it computes all valid reordered data flows with the help of an example data flow $D = [\texttt{Src} \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3]$. The flow consists of a data source `Src` and three Map operators with $\texttt{Map}_3$ being the root. We assume that all Map operator pairs can be reordered except for $\texttt{Map}_2$ and $\texttt{Map}_3$. The algorithm starts by recursively enumerating all reordered alternatives $Alts_{-r}$ for D_{-r}, which is the input data flow D minus the root operator r ($\texttt{Map}_3$) (Line 18):

$$\begin{aligned} Alts_{-\texttt{Map}_3} &= \texttt{Enum-Alternatives}([\texttt{Src} \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_2]) \\ &= \{[\texttt{Src} \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_2], [\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_1]\} \end{aligned}$$

The result of the first recursive call $Alts_{-r}$ is used for two purposes. First, to enumerate a subset of the result $Alts$, namely all reordered flows with the original root r. This is achieved by simply appending the root r ($\texttt{Map}_3$) to each computed alternative $A_{-r} \in Alts_{-r}$ (Line 21):

$$\begin{aligned} Alts = \{&[\texttt{Src} \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3]\} \cup \\ &\{[\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_3]\} \end{aligned}$$

Second, $Alts_{-r}$ is used to retrieve candidate root operators s that can be reordered with r. For each root s of the computed alternatives $A_{-r} \in Alts_{-r}$, the algorithm checks whether it can be

Algorithm 4 Enumeration of alternative data flows

```
1:  function ENUM-ALTERNATIVES(D)
2:      input: data flow D
3:      output: all possible data flows derived by reordering of D
4:      Alts = mTab.get(getMTabKey(D)) // check memoTable
5:      if (Alts ≠ ∅) then
6:          return Alts
7:      r = getRoot(D) // get root r of D
8:      if (r is data source) then
9:          Alts = {r}
10:     else if (r is data sink) then
11:         D_-r = rmRoot(D)
12:         Alts_-r = Enum-Alternatives(D_-r)
13:         for (A_-r ∈ Alts_-r) do // add r to each A_-r
14:             Alts = Alts ∪ {addRoot(A_-r, r)}
15:     else if (r is single-input operator) then
16:         cand = ∅
17:         D_-r = rmRoot(D)
18:         Alts_-r = Enum-Alternatives(D_-r)
19:         for (A_-r ∈ Alts_-r) do
20:             s =getRoot(A_-r) // get candidate root s
21:             Alts = Alts ∪ {addRoot(A_-r, r)} // add r to A_-r
22:             if (s ∉ cand ∧ reorderable(r, s)) then
23:                 cand = cand ∪ {s} // enum candidate s only once
24:                 D_-s =setRoot(A_-r, r) // replace s by r
25:                 Alts_-s = Enum-Alternatives(D_-s)
26:                 for A_-s ∈ Alts_-s do // append s to each A_-s
27:                     Alts = Alts ∪ {addRoot(A_-s, s)}
28:     mTab.put(getMTabKey(D), Alts)
29:     return Alts
```

reordered with the original root r ($\texttt{Map}_3$) by calling the Boolean function $\texttt{reorderable}(r,s)$ (Line 22). In our example, this is only true for $s = \texttt{Map}_1$ and $r = \texttt{Map}_3$ since $\texttt{Map}_3$ and $\texttt{Map}_2$ cannot be reordered. Therefore, $\texttt{Map}_3$ replaces $\texttt{Map}_1$ as root of $A_{-r} = [\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_1]$, i. e., r is pushed down to data flow D_{-s} (Line 24):

$$D_{-\texttt{Map}_1} = [\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3]$$

The successive recursive call $\texttt{Enum-Alternatives}(D_{-s})$ enumerates all valid reorderings for the D_{-s} (Line 25):

$$\begin{aligned} Alts_{-\texttt{Map}_1} &= \texttt{Enum-Alternatives}([\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3]) \\ &= \{[\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3]\} \end{aligned}$$

The result set $Alts$ is amended by all valid reorderings that have s as root. This is achieved by

simply appending s to all reordered flows $A_{-s} \in Alts_{-s}$ (Line 27):

$$Alts = Alts \cup \{[\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3 \rightarrow \texttt{Map}_1]\}$$

Finally, all computed alternatives $Alts$ are returned:

$$\begin{aligned} Alts = \{&[\texttt{Src} \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3], \\ &[\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_1 \rightarrow \texttt{Map}_3], \\ &[\texttt{Src} \rightarrow \texttt{Map}_2 \rightarrow \texttt{Map}_3 \rightarrow \texttt{Map}_1]\} \end{aligned}$$

In order to avoid duplicate enumerations, the algorithm may only descent once into recursion for each distinct root candidate s (Lines 16, 22, 23). The use of a memo table reduces the number of recursive descents and improves the enumeration time (Lines 4, 28).

The enumeration algorithm can also be easily integrated with a Volcano-style physical optimizer using interesting properties as described in [23, 97]. Instead of computing and returning all valid reordered data flows, the `Enum-Alternatives()` function can be adapted to compute the least expensive physical execution plan for each interesting property. Additionally, the algorithm must take care that at least one plan for each possible root node s of a sub-flow is returned, in order to enumerate all possible reorderings. Physical execution plans are generated by recursively computing the least expensive execution plans for sub-flows, choosing local and shipping strategies only for the root node, and connecting it to the sub-plan. Interesting properties can be tracked during recursive descent and be used to enumerate physical execution plans for sub-flows. By integrating cost-based physical optimization in the enumeration algorithm, the principle of optimality can be exploited which effectively reduces the number of enumerated alternatives.

In contrast to optimization of relational queries, our approach for enumerating reordered data flows is limited by the choice of the initial data flow. For some queries, such as queries that include circular join graphs, the initial data flow already implies a plan decision that cannot be changed by reordering operators.

4.5 Evaluation

We implemented a prototype to evaluate our approach for data flow optimization. The prototype is based on the Stratosphere system, which is available as open source [195][2]. Furthermore, we implemented data processing tasks from different domains as PACT programs to experimentally evaluate and validate our approach. These include relational OLAP, as well as weblog clickstream processing and biomedical text mining. Our experimental evaluation covers the following aspects. First, we assess the optimization potential for parallel data flows. Second, we evaluate the plan space enumerated by our optimizer. Third, we discuss the overhead of the plan enumeration algorithm. Finally, we verify that static code analysis can be used to derive the necessary properties for reordering UDFs.

We start discussing our prototypical implementation and present the PACT programs used for evaluation before we show and discuss experimental results.

4.5.1 Experimental Setup

The existing optimizer of Stratosphere performs cost-based physical optimization as known from parallel relational optimizers, i. e., it selects data shipping and execution strategies, such as broadcasting and hybrid-hash joins, for a given data flow [23]. The cost model is a combination of network I/O, disk I/O, and CPU costs of UDF calls. For result size and cost estimations, the optimizer relies on hints, such as *"Average Number of Records Emitted per UDF Call"*, *"CPU Cost per UDF Call"*, and *"Number of Distinct Values per Key-Set"*. These can be provided by the user, a language compiler (e. g., Hive or Pig), or obtained by runtime profiling.

In order to implement our prototype, we adapted the optimization process of Stratosphere's optimizer in the following ways. Prior to plan enumeration, the optimizer obtains information about the UDFs which is required to reason about reorderability of operators. This information can be provided by manually attached annotations or derived by an SCA component. Our SCA component is based on the Soot framework [189], which provides all features required by our code analysis technique (see Section 4.3). It also takes care of establishing the global record. After the information has been obtained, all valid alternative data flows are computed using the enumeration algorithm presented in Section 4.4. The existing cost-based optimizer [23] is called for each alternative to choose shipping and local strategies and compute a cost estimate. Finally, the cheapest plan is selected and returned for execution.

We perform our experiments on a cluster of four machines each being equipped with two Intel Xeon E5530 Quadcore CPUs, 48 GB RAM, and ten 250 GB disks for data bundled in a RAID5. The machines are connected with 1 GBit Ethernet and run Linux (Ubuntu Server 10.04.3 LTS), Sun Java 6, and HDFS 0.20.2. We execute all tasks with a degree of parallelization of 32.

[2] Stratosphere evolved into Apache Flink [84].

4.5.2 Evaluation Programs

We evaluate our approach using four tasks from different domains. Algebraic optimization of relational queries is best known from relational DBMS but also applied in the context of parallel data flow systems by higher-level languages, such as Hive [199], SCOPE [47], and Tenzing [49]. In order to show the effectiveness of our approach, we implemented two queries of the TPC-H benchmark for evaluation. Parallel data flow engines are commonly used for non-relational tasks. We show the applicability of data flow optimization for such domains by providing two non-relational tasks, namely biomedical text mining and weblog clickstream processing. All four tasks are implemented as handcrafted PACT data flows. In this section, we shortly present all tasks and their implementations.

Relational OLAP We implemented slightly modified variants of queries 7 (where we reduced the selectivity of the `shipdate` filter and removed the final sorting) and 15 (where we removed the filter on `total_revenue`) from the TPC-H benchmark to cover relational analytical tasks. For our experiments, we run both queries on a 400 GB TPC-H data set. Query 7 applies a local predicate on the lineitem relation, joins six relations with circular-connected join predicates, and finally performs a grouping with sum aggregation. Figure 4.5(a) shows our PACT implementation. All joins are implemented as Match operators except the join with the disjunctive join predicate ($\texttt{nation}_1 \bowtie \texttt{nation}_2$), which is implemented as a filtering Map operator. Grouping and sum aggregation are realized as a Reduce operator. Query 7 was chosen to demonstrate the join reordering capabilities of our approach.

Our query 15 applies a local predicate on the lineitem relation, joins it with the supplier table, and groups and aggregates to compute the final result. We implemented the local predicates as a Map, the join as a Match, and the grouping and aggregation as a Reduce operator (see Figure 4.6(a)). Note that the join predicate and the grouping clause reference the same attribute (`s_key`). As shown in Section 4.2.3, this is a necessary condition to reorder a Match and a Reduce operator. We selected query 15 as an example to showcase that our approach is able to push an aggregation past a join.

Text Mining We implemented a text mining task that detects relationships between genes and drugs described in biomedical text corpora. The data flow is a pipeline of Map operators, which extract entities and relationships by applying several natural language processing (NLP) algorithms to the input. Our program takes a text collection as input and performs some linguistic preprocessing, e. g., tokenization and part-of-speech tagging on the input to enable entity and relation extraction. In order to reduce intermediate result set sizes, each entity or relation extraction component also works as a filter by forwarding only those records that actually contain a gene, drug, or relation mention. Most NLP components are compute-intensive and call third-party machine learning or automaton-based libraries. Furthermore, most components have

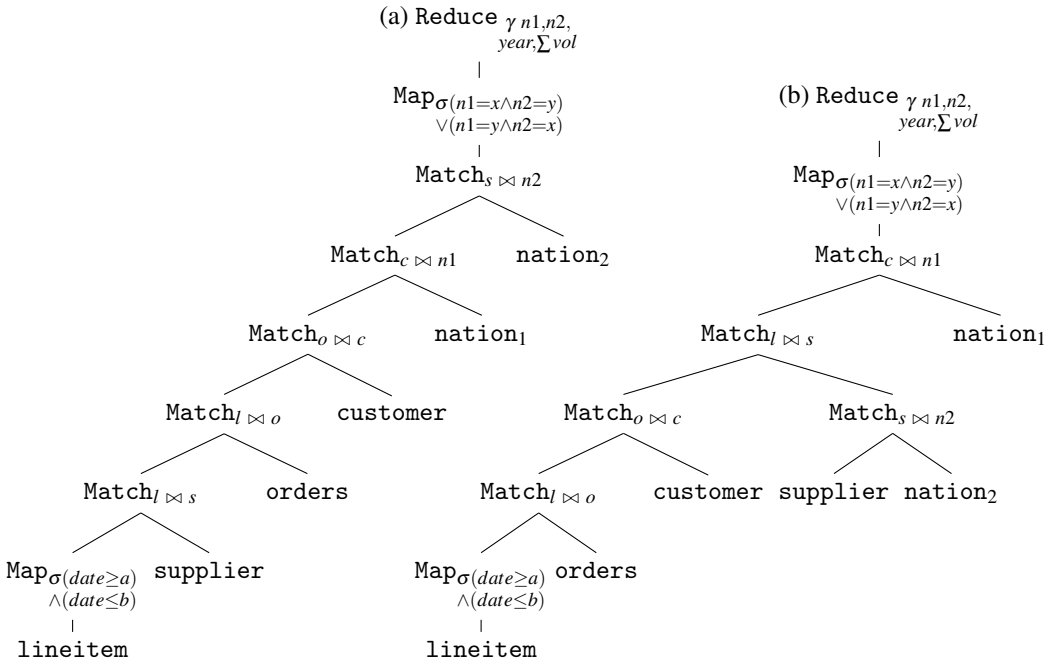


Figure 4.5: PACT data flows of Query 7: (a) Implemented data flow, (b) 1st-ranked reordered data flow.

dependencies on other components to be executed in advance. These dependencies limit the set of valid reordered data flows. Optimization potential arises from different filter selectivities and varying execution costs for the text mining components. We execute the text mining data flow on a 425 MB subset of the PubMed data set.

Clickstream Processing Weblog processing is a common example of non-relational data flows [87]. We implemented a task that processes web shop clickstream data (see Figure 4.7(a)). The task extracts click sessions that lead to buy actions and augments them with detailed user information. Such tasks are common preprocessing steps for data mining algorithms. In our scenario, a clickstream entry contains an IP address, a timestamp, and a visited URL. The URL encodes a session id, and the performed user action. The first Reduce operator filters on sessions that include at least one buy action. The successive Reduce operator condenses a session into a single record. The following Match operator joins by session id with a relation that resolves session ids to ids of logged in users, thereby selecting only sessions with logged in users. Finally, a second Match operator appends detailed user information by joining on the user id. For our experiments, we ran the task on 430 GB `click`, 13.8 GB `login`, and 9.2 GB `user_info` data.

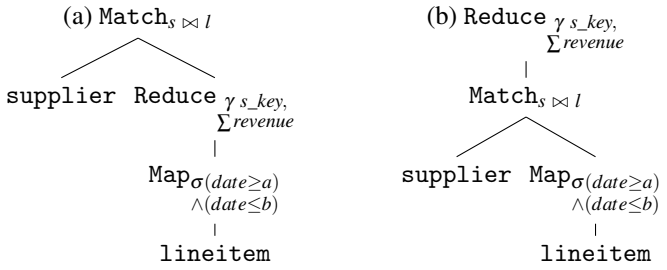


Figure 4.6: PACT data flows of Query 15: (a) Implemented data flow, (b) Data flow with Match before Reduce.

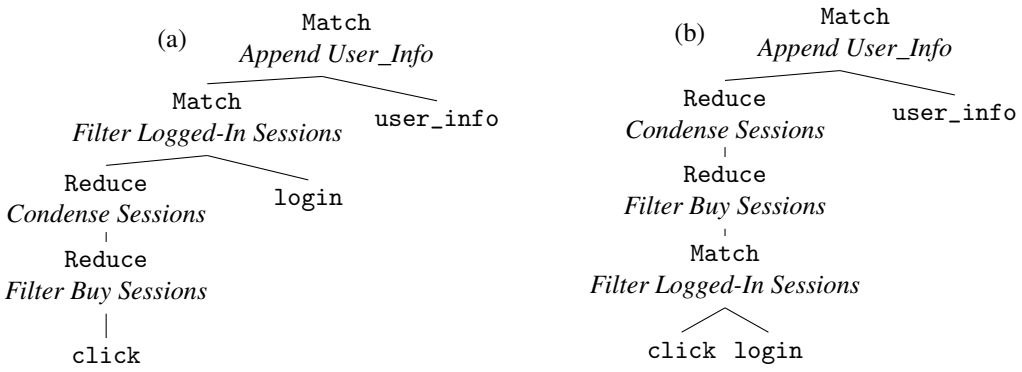


Figure 4.7: PACT data flows of clickstream processing task: (a) Implemented data flow, (b) 1st-ranked reordered data flow.

4.5.3 Experiments

Our evaluation addresses different aspects of our approach. First, we assess the potential of optimizing data flows with user-defined functions by comparing the execution time of equivalent data flows with reordered operators. Second, we analyze the space of plan choices that is covered by our reorder conditions. Third, we briefly discuss the overhead of enumerating reordered plan alternatives, and finally conclude our analysis by assessing the usefulness of our static code analysis approach to extract the properties, which are required to evaluate our reordering conditions.

Optimization Potential Query optimization as performed by modern relational DBMSs has the potential to improve query execution time by orders of magnitude. Our first set of experiments assesses the potential of our generic data flow optimization technique. We enumerate all possible data flows for a given PACT program. Each reordered alternative is fed into the physical optimizer where shipping and local execution strategies are enumerated, and cost estimates are obtained. We sort the resulting plans in ascending order by their estimated costs and assign a

rank to each plan that corresponds to its position in the list. We pick ten plans in regular rank intervals from the list and execute them. For each executed plan, we plot the cost estimate of the optimizer and the actual execution time (averaged over three runs), both normalized by the lowest estimated costs and averaged execution time respectively.

Figure 4.8 shows the results for TPC-H query 7. The enumeration algorithm explored a space of 2518 alternative plans. We see that the plan with the least estimated costs provides also the least execution time with an absolute execution time of roughly 6 minutes (see Figure 4.5(b) for this plan). The last ranked plan is slower by a factor of 7 and requires the most time for execution (about 45 minutes).

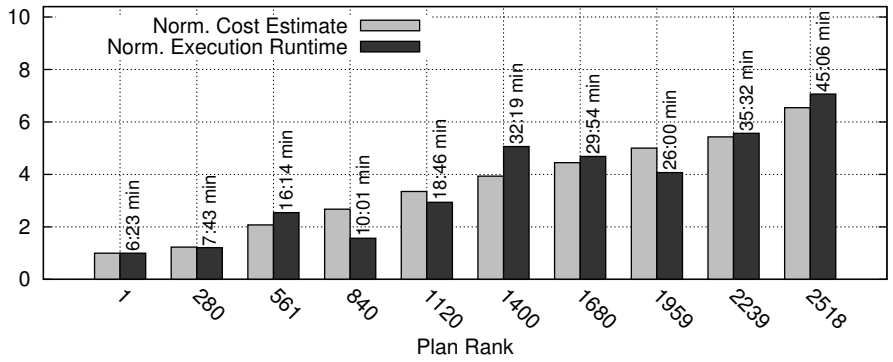


Figure 4.8: Normalized cost estimates and execution time for 10 regularly picked execution plans of the TPC-H Query 7.

Figure 4.9 shows the estimated costs and execution time for selected plans of the text mining task. The best plan (according to estimated costs) outperforms the worst by almost an order of magnitude.

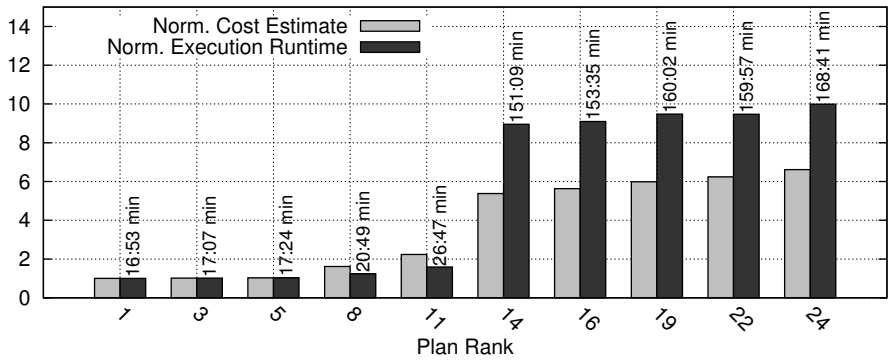


Figure 4.9: Normalized cost estimates and execution time for 10 regularly picked execution plans of the text mining job.

Our experiments show that reordering of data flows can lead to significant performance improvements. Due to the observation that in general execution plans with higher cost estimates require

more time for execution, we can also approve the validity of the optimizer's cost model. We note that Stratosphere does not support indexes, columnar layouts, or materialized views yet. Therefore, all execution plans result in full scans of all included data sets, which limits the achievable performance improvements.

Plan Enumeration Space We continue discussing the plan enumeration space with TPC-H query 15. Our implementation is based on a Map, a Reduce, and a Match operator (see Figure 4.6). We can exchange Match and Reduce since the ROC condition is fulfilled, Match preserves the group cardinality because `s_key` is unique in the supplier relation[3], and Reduce groups on the Match key (`s_key`). This is essentially an aggregation push-up rewrite that could also be applied by a relational optimizer. Besides the changed order of Reduce and Match, the rewrite also leads to different physical plan choices.

For the data flow with Reduce being the input of Match (Figure 4.6(a)), the physical optimizer chooses to partition the input of Reduce and establish the groups by sorting. The grouped and aggregated result is locally forwarded into the Match operator and used to build a hash table. Since Match operates on the same key as Reduce, the partitioning property remains and can be reused. To compute the final result, the supplier relation is also partitioned, shipped to the Match operator, and probed against the hash table. In fact, the optimizer could also choose to reuse the sorting of Reduce and perform a sort-merge join for Match. However, this would require to sort the supplier relation.

The alternative data flow with Match being the input of Reduce (Figure 4.6(b)) is executed using a different shipping strategy. In this case, Match's lineitem input is much larger than the supplier input, since it has not been aggregated as in the previous case. Therefore, the optimizer decides to broadcast the much smaller supplier input to all parallel instances of Match and build a hash table from it. The lineitem side is locally forwarded and probed against the hash table. The result is partitioned and shipped to Reduce, which groups by sorting and computes the final result.

As previously stated, our optimizer is also able to reorder non-relational operators. Figure 4.7 shows (a) the implemented PACT program and (b) the data flow chosen by the optimizer for the clickstream processing task. Both Reduce operators are non-relational operators. The *"Filter Buy Sessions"* UDF is called with all click records of a session and checks whether at least one click performs a buy action. In that case, all click records are forwarded, otherwise none. The subsequent *"Condense Sessions"* UDF collects all clicks of a session, merges them into a single record and forwards it. Comparing the best performing and the implemented data flow, we see that the optimizer pushed the selective join (*"Filter Logged-In Sessions"*) below both non-relational Reduce operations. We are not aware of a data processing system that is able to perform similar optimizations. Figure 4.10 gives the estimated costs and execution times of all

[3] We used a hint to tell the PACT optimizer about this uniqueness.

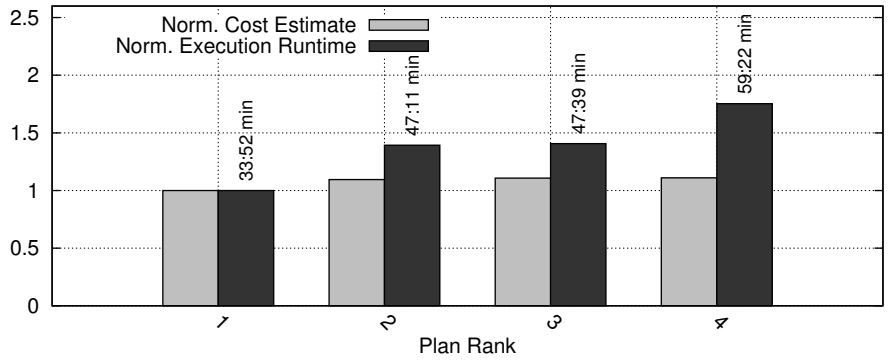


Figure 4.10: Normalized cost estimates and execution time for all 4 execution plans of the click-stream processing job.

four execution plans. The best performing plan beats the implemented data flow (Rank 3) by a factor of 1.4 or 13:47 minutes.

Our optimizer explores large fractions of the search space that conventional relational optimizers cover, including bushy join orders (Figure 4.5), pushed aggregations (Figure 4.6), and reasoning about interesting properties [23]. Furthermore, we show that our approach enables optimizations that are not supported by any current data analysis system we are aware of (Figure 4.7).

Enumeration Time Our enumeration algorithm is facing the same problem of exponential search space sizes as relational optimizers. As previously discussed, our prototypical implementation first enumerates all valid reordered data flows and subsequently calls the physical optimizer for each candidate. This implementation does not permit cost-based search space pruning and it is not tailored towards efficient plan enumeration. In Section 4.4 we gave an intuition of how to integrate the enumeration algorithm with the physical optimization step. For all queries presented so far, which represent typical data analysis tasks, plan enumeration took less than 1.7 seconds using our naive implementation. The overhead of performing the static code analysis is virtually zero.

Feasibility of Static Code Analysis We evaluate the feasibility of static code analysis to determine read and write sets of UDFs. For this purpose, we compare the number of reordered alternative data flows that were enumerated based on read and write sets which were manually annotated and automatically derived using static code analysis. Table 4.2 gives the results for all presented evaluation tasks. The information extracted by our prototypical implementation of the SCA component enables the optimizer to enumerate almost all valid plans for our four evaluation data flows. The current implementation is restricted to information that is available at UDF compile time and can be easily accessed, such as field accesses with literals and final

PACT Task	Enumerated Orders with Manual Annotation	Enumerated Orders with SCA
Clickstream	4	3 (75%)
TPC-H Q7	2518	2518 (100%)
TPC-H Q15	4	4 (100%)
Text Mining	24	24 (100%)

Table 4.2: Comparing number of reordered alternatives for manually annotated and automatically derived read and write sets.

variables. This can be extended to more exhaustive control flow tracking and incorporation of job configurations, which are only available at optimization time.

4.6 Related Work

There is a good amount of prior work that is related to certain aspects of our approach. However, we are not aware of any work which aims to optimize data flows with unknown operator semantics by reordering.

Optimization of user-defined functions and predicates has been extensively discussed in the context of extensible RDBMSs [53, 118, 119]. This line of work only considered UDFs with strict templates that revealed enough information about the semantics, such as user-defined predicates or scalar functions. Hence, the challenge was not to identify whether a UDF can be reordered, but rather when it is beneficial. Techniques to address this problem included estimation and monitoring of UDF execution cost and selectivity.

Section 2.2.3 gave an overview of the optimization of higher-level languages, such as Pig [169], Jaql [28], Hive [199], DryadLINQ [206], SCOPE [47], AQL [25], and Tenzing [49]. These approaches treat UDFs as fixed blocks in data flows and include only operators with known semantics into their optimization. As a consequence, operators cannot be reordered across UDFs. In contrast to these approaches, our work applies optimization directly to data flows without knowledge of the operator's algebraic properties. We view the two approaches as complementary; while we show that some optimizations can be performed at the data flow level, thus making a data flow engine able to seamlessly handle multiple data and programming models, other optimizations are semantic in nature and can only be performed at a higher level. We note that higher-level language translators can enrich data flows with reordering information based on the operator's semantics, hence enabling the unified optimization of operator order and physical optimization at the data flow abstraction. Rheinländer et al. [180] presented an extensible optimizer for data flows with user-defined operators as a follow-up of our work. The approach is based on an extensible taxonomy of semantic operator annotations and set of transformation rules. Read and write sets are automatically inferred and attached as semantic annotations to operators.

Starfish [121] and Stubby [151] are approaches to optimize the execution of MapReduce programs and workflows. In contrast to our work, Starfish does neither inspect nor optimize the program itself. Instead, it uses runtime profiling, execution simulation, and cost-based optimization techniques to generate well-performing job configurations for Hadoop MapReduce jobs. Stubby relies on manual code annotations to optimize workflows of multiple Hadoop MapReduce programs by reducing the number of executed MapReduce programs. This is achieved by merging subsequent programs (vertical merging) or programs that operate on the same input data (horizontal merging). Among the relevant information for Stubby are modifications of key fields. In principle, this information could also be gathered using static code analysis under certain conditions.

There are a few approaches that employ static code analysis for data flow optimization. Manimal [136] applies static code analysis to MapReduce programs to identify relational-style selec-

tions and projections. An optimizer selects from available B^+-tree indexes and decides on the use of delta-compression. Manimal's optimizations are orthogonal to ours (operator reordering), and would constitute valuable additions to our system. Scope analyzes user-defined functions to infer whether UDFs preserve existing data properties, such as partitioning or sorting [211]. In fact, this analysis could also be performed given the write sets of our approach. Scope goes even beyond non-invasive analysis and modifies the code of user-defined function to push down filters and projections [106]. None of these approaches leverages static code analysis to reason about reorderbility of UDFs.

The concept of read and write sets to detect conflicting data accesses is similar to conflict detection in optimistic concurrency control methods [144] and compiler techniques to optimize loops. Finally, we draw inspiration from the Ferry project [104]. Ferry follows an algebraic approach to push data processing instructions from the application into the DBMS by translating general-purpose (application) code to SQL queries.

4.7 Summary

In this chapter, we proposed and addressed the problem of optimizing data flows that consist of black box user-defined functions written in an imperative language. In this setting, the algebraic properties of the operators of the data flow are unknown, and must be discovered. Our key insight is that a handful of properties, which can be discovered using static code analysis, suffice to establish many optimizations known from relational algebra, including filter and join reordering, and some forms of aggregation push-down. We formally establish reordering conditions, show how to estimate the desired properties via static code analysis, and present a plan enumeration algorithm.

We have prototyped our solution in the Stratosphere system. Our experimental results show that our approach is able to reorder relational and non-relational data flows, leading to performance improvements of up to an order of magnitude. Moreover, we demonstrate that our approach is able to perform optimizations which algebraic optimizers are not capable of. Our experiments attest, that our static code analyzer successfully extracts properties from black box UDFs that are required for reordering them.

5 Assessing the Risk of Relational Data Flows

Contents

Cost-based query optimization is the standard approach to choose an execution plan for a relational query from all possible semantically equivalent execution plans. Since the performance of these plans can vary by several orders of magnitude, the quality of plan choices is of utmost importance for the overall performance of a database system. Cost-based optimizers estimate the execution cost of candidate plans and choose the plan with the least estimated cost. When estimating the cost of a plan, the optimizer assesses the conditions under which the query will be executed and calculates the cost based on its cost model parameterized with these conditions. Important cost model parameters are, among others, the amount of data to process and the size of the available memory budget. A major problem of this approach is the fact that the execution conditions of a query are not known at optimization time and must be predicted. The quality of an optimizer's plan choice depends on its ability to assess the execution conditions of a plan when it is executed. While some of the execution conditions, such as the costs of I/O operations, can be obtained by performance profiling and monitoring, others are notoriously hard to assess. These include the cardinality of intermediate results and the amount of available resources. As a consequence, cost-based optimizers base their plan choices on potentially false assumptions.

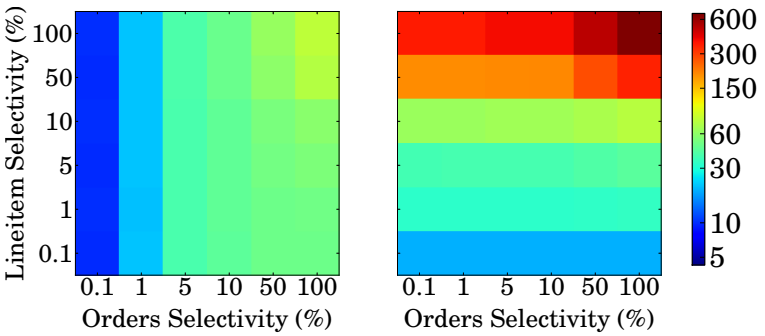


Figure 5.1: Execution time (s) of two semantically equivalent plans with two predicates of varying selectivity.

While it is not surprising that the performance of a plan depends on parameters such as memory budget or input cardinalities it is important to note that not all plans behave identically for changing execution conditions. Figure 5.1 shows the execution time of two semantically equivalent plans for a version of TPC-H query Q3 [200], which was taken from the Picasso project [174]. This query has two parameterized local range predicates on `lineitem` and `orders` for which the selectivities have been varied in steps 0.1%, 1%, 5%, 10%, 50%, and 100%. While the left plan has nearly constant performance for variations of the `lineitem` predicate and an execution time ranging from 10 to 80 seconds, the right plan shows a different pattern. There, varying the selectivity of the `lineitem` predicate causes the execution time to vary from about 20 to 600 seconds. Less variation is caused when changing the selectivity of the `orders` predicate. While the performance of the right plan varies by a factor of 30, the performance of the left plan only varies by a factor of 8. The above example illustrates that plans can differ in their sensitivity for changing execution conditions. We call plans with rather slightly or moderately varying execution time for changing execution conditions *robust* plans and plans with significantly changing performance *risky* plans.

Risky plans pose a serious problem for systems that use cost-based query optimizers. If the optimizer is not able to accurately predict the conditions in which a plan will be executed, the cost model is called with imprecise parameters and the optimizer will pick a suboptimal plan. If a robust plan with low performance variation is chosen, this might not be an issue. However, a plan which is sensitive to the size of its main memory budget can exhibit an unexpected execution time if it receives less memory than expected due to concurrently running queries. Similarly, the performance of a plan that was chosen because it is efficient for small data sizes might degrade if it has to process more data than expected. Because risky plans often benefit from underestimated input sizes they are frequently preferred over more robust plans with higher fixed costs. Hence, risky plans are often chosen and executed where more robust plans would have been more efficient.

Risky execution plans and the challenges of predicting execution conditions are not RDBMS-specific problems. All systems that employ a cost-based optimizer are affected by this issue. In fact, the problem of accurately estimating execution conditions is even harder in the context of most optimization approaches discussed in Chapter 2.2. The presence of user-defined functions, data without schema and statistical information, heterogeneous and distributed compute nodes, virtualization, and concurrently running applications, renders the estimation of execution conditions even more challenging if not impossible in certain cases.

In this chapter we discuss three important aspects of plan risk in detail.

1. We analyze the impact of changing execution conditions on the performance of execution plans and identify risky plan patterns.
2. We propose a metric to measure plan risk and show how plan risk information can be leveraged to improve optimizer plan choices.
3. We propose an approach to predict the risk of execution plans.

To limit the number of parameters that influence the performance of execution plans and to ease the analysis of the observed effects, we restrict the scope of this chapter to execution plans for stand-alone relational database systems.

The remainder of this chapter is organized as follows. In Section 5.1 we analyze the behavior of execution plans for changing execution conditions, identify plan patterns that can cause significantly varying execution performance, and derive a better understanding for plan robustness. We start studying the performance of individual operators and common plan patterns for changing execution conditions. Subsequently, we report on an extensive experimental study to assess the execution behavior of 306 plans for changing input sizes and memory budgets and analyze its results. In Section 5.2 we propose a risk score metric to measure the riskiness of execution plans and show how it can be used to prevent the choice of plans with extremely varying performance. In Section 5.3 we present an approach to predict the risk score of query execution plans using a machine-learned regression model. The model is trained with data obtained from the previously presented study. We give an overview of related work in Section 5.4 and summarize the chapter in Section 5.5.

The work presented in this chapter was inspired by discussions at the Dagstuhl seminar 12321 on “Robust Query Processing” in August 2012. We thank Surajit Chaudhuri, Goetz Graefe, Ihab F. Ilyas, Anisoara Nica, Meikel Poess, and Kenneth Salem for great discussions.

5.1 Analyzing the Performance of Query Plans for Changing Execution Conditions

Cost-based query optimizers choose execution plans by comparing their estimated execution costs. In order to obtain meaningful cost estimates, optimizers must predict the conditions in which a query will be executed. However, many relevant execution conditions cannot be precisely assessed at optimization time and need to be estimated. A major issue of cost-based optimization is that query execution plans can differ considerably in their sensitivity when execution conditions change. In the case of misestimated execution conditions, a chosen plan can perform significantly worse than expected. Among the most important parameters for estimating the cost of query execution plans which are often not known at optimization time are the size of the plan's memory budget and the cardinality of its intermediate results.

Disk-based database systems try reduce the number of physical I/O operations, which account for a large portion of query execution costs, by caching data in memory. Therefore, the size of a query's memory budget is an important cost model parameter. However, memory is a resource that is shared among concurrently running queries and operators. Hence, the amount of memory that is available for a query frequently changes depending on the number and resource requirements of other concurrent queries. Similar reasoning applies to the amount of available CPU and I/O resources, which are also shared among concurrently running queries or even external processes on the same physical infrastructure, as in the case of virtualized environments. An optimizer would need to look into the future to accurately estimate the amount of available resources and the number of concurrently running queries for the time when the plan is executed.

The size of the input data of an operator is another crucial parameter when estimating its cost. It can be computed from the size of an input tuple and the number of input tuples (cardinality). While the size of a tuple can be easily estimated from schema information and statistics, cardinality estimation of intermediate results is significantly harder and requires detailed knowledge about the data to be processed. In a nutshell, the resulting cardinality of a (sub)plan is estimated from the cardinalities of all involved base relations and the selectivities of all involved operators. Since base cardinalities can be obtained and maintained with low overhead, this information is usually available and quite accurate. However, estimating the selectivity of operators is more intricate. Therefore, the problem of input size estimation essentially comes down to selectivity estimation.

Database systems gather statistical information about the data they store in order to support selectivity estimation. Since storage and maintenance of statistics requires space and incurs overhead, it is not feasible to collect detailed statistics for all base relations and attributes. Instead, database administrators carefully choose the type and level of detail of statistics to collect. Statistics are typically maintained periodically since online maintenance is too expensive. Due

to the approximate nature of statistics, statistical information cannot provide exact selectivity estimates such that optimizers need to approximate selectivities from the available information. Conventional optimizers follow a number of heuristic assumptions for this purpose, such as the uniformity assumption, the independence assumption, and inclusion assumption. Especially the independence assumption is commonly violated by real-world applications and prone to cause severe cardinality underestimation [177]. In some cases where no statistical information is available, optimizers fall back to default values for selectivity estimates. Due to lack or inaccuracy of available statistics and the use of default assumptions and values, selectivity estimates can be off by multiple orders of magnitude [131, 164] in practice.

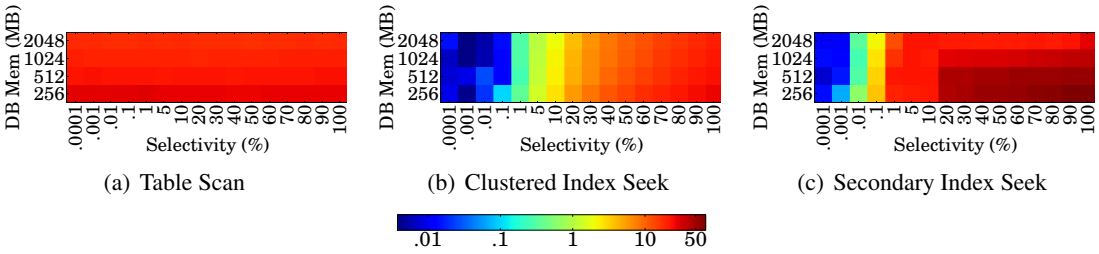
The general problem of selectivity estimation applies to all kinds of queries. For parameterized queries additional aspects impede the assessment of execution conditions. Queries with parameters are often optimized once and the resulting plan is used for multiple executions of the same query with different parameter bindings. Hence, the size of intermediate results can change significantly for different parameter bindings, the size or the properties of the queried data might change between two runs of the plan, and the amount of available resources might change as well over time.

In this section we experimentally measure and analyze the performance of query execution plans for varying execution conditions. We gain a better understanding of what differentiates plans that are sensitive and insensitive for changing execution conditions and identify plan features that can cause significantly varying execution times. Our analysis focuses on the impact of varying input sizes and memory budgets, as these parameters are hard to estimate for optimizers, have a large impact in the performance of a plan, and are rather easy to control in an experimental setting. We start analyzing the execution behavior of individual operators (Section 5.1.1) and common plan patterns (Section 5.1.2). Subsequently, we present the results of an extensive experimental study to assess the risk of query execution plans (Section 5.1.3) and conclude with a list of risky plan features (Section 5.1.4).

5.1.1 Performance of Operators

Query execution plans consist of physical operators, such as `TableScan`, `HybridHashJoin`, and `Sort`. The behavior of a plan for changing input sizes or memory budgets depends on the behavior of its individual operators. In this section, we briefly discuss the performance characteristics of selected operators for varying execution conditions.

Accessing Relations Relation access operators are often responsible for a large fraction of a plan's execution costs in disk-based DBMS. Most database systems support at least three different strategies to read and filter a relation, namely 1) a full `TableScan` with a subsequent `Filter`,



(a) Table Scan (b) Clustered Index Seek (c) Secondary Index Seek

Figure 5.2: Execution time (s) of single predicate table access plans for varying predicate selectivity and database memory

2) a `ClusteredIndexSeek`, and 3) a `SecondaryIndexSeek`, given that the corresponding indexes exist.

The `TableScan` operator sequentially reads all disk pages of a relation and evaluates filter predicates on each individual tuple. Indexes can help to reduce the number of I/O operations, such that only disk pages with relevant tuples are read from disk. A clustered index stores the full relation ordered on the indexed attribute with additional pointers into the sorted sequence. A `ClusteredIndexSeek` operator seeks to the first tuple that matches the predicate and sequentially reads tuples as long as the predicate evaluates to true. In contrast, a secondary index only holds pointers to its base relation, which can be in any order. Therefore, a `SecondaryIndexSeek` causes random I/O accesses when retrieving data.

Figure 5.2 shows the execution time of these three operators for accessing a 1 GB table with 1 million records and applying a range predicate with selectivities from 0.0001% to 100% and the total memory budget of the DBMS varying from 256 GB to 2048 MB. Since the I/O behavior of the `TableScan` operator does not depend on the selectivity of the predicate, it has a uniform performance (Figure 5.2(a)). Its execution time differs only slightly for changes of the database memory, which might be due to overhead caused by page evictions of the buffer pool. In contrast, the execution costs of the index seek operators depend on the selectivity of the filter predicate. The `ClusteredIndexScan` provides low execution times for small selectivities and degrades gracefully for increasing selectivities (Figure 5.2(c)). For a selectivity of 100% it is as good as the `TableScan` operator because it also reads all pages of the table in sequential order. Similar to the `TableScan` plan, the performance does only marginally depend on the amount of available memory. The `SecondaryIndexScan` plan provides good performance for low predicate selectivities of up to 0.1% (Figure 5.2(b)). However, its execution time significantly increases for selectivities larger than 1% due to its random I/O access pattern. Furthermore, its performance depends on the size of the buffer pool and hence on the amount of DBMS memory. If the whole relation fits into the buffer pool, each table page can be cached in the buffer pool and subsequent reads of the same page are served from memory without accessing the disk. We can assume that for 2048 MB DBMS memory most of the 1 GB table fits into the buffer pool such that the performance of the secondary index plan is close to the performance of the `TableScan`

plan. Note that even in case of a large buffer pool, the performance of `SecondaryIndexSeek` operators can suffer from concurrently running operators and queries which replace disk pages in the buffer pool.

Joining Most analytical queries include joins. In addition to the order of join operators, the type of join operators can have a significant impact on the performance of a query execution plan. Most relational database systems support at least four types of equality joins, `MergeJoin`, `NestedLoopJoin`, `IndexNestedLoopJoin`, and `HybridHashJoin`.

A `MergeJoin` expects its inputs to be sorted on the join attributes. Both inputs are consumed in a zig-zag fashion, always advancing the input with the smaller join attribute value (assuming ascending order). In case of `1:1` or `1:n` joins, a `MergeJoin` does not hold any intermediate data and produces its (sorted) result as a pipelined stream. Hence, its performance depends linearly on the size of its larger input.

The `NestedLoopJoin` is the most generic join algorithm. It can evaluate arbitrary join conditions and does not have any requirements on its inputs. The inputs are distinguished as *outer input* and *inner input*. In its basic form, the `NestedLoopJoin` repeatedly reads its inner input for every tuple of the outer input and applies the join predicate to each pair of outer and inner tuple. More efficient implementations iterate over the inner input for a batch of outer tuples. While being fairly efficient for small inputs, the performance of a `NestedLoopJoin` drastically degrades for increasing cardinality of the outer side because more and more iterations over the inner input are required.

The `IndexNestedLoopJoin` is a common special case of the `NestedLoopJoin` where the inner input is a `SecondaryIndexSeek` or `ClusteredIndexSeek` operator. For each tuple of the outer input, an index seek operation is performed to retrieve matching tuples from the index. The `IndexNestedLoopJoin` provides good performance for small outer and large indexed inner sides because the inner side is not completely read but only matching tuples are fetched from the index. However, similar to the plain `NestedLoopJoin`, the cost of an `IndexNestedLoopJoin` significantly increases for growing cardinalities of the outer input as more random I/O operations are caused by index fetches. The impact of varying outer sides is reduced under certain conditions. For example, the number of random physical I/O operations decreases if the inner relation fits to a large extent into the buffer pool or if the outer input is sorted on the join attribute and the inner input is a clustered index. In the latter case, the disk pages of the clustered index are sequentially read (and might even be pre-fetched), which reduces the I/O cost significantly.

The `HybridHashJoin` has a *build* and a *probe* input and works in two phases. In the build phase, all tuples of the build input are inserted into a hash table, which is indexed by the join attribute. In the subsequent probe phase, the hash table is probed with the join attribute of each tuple from the probe input and matching tuples are joined. The `HybridHashJoin` excels if the hash table can be kept completely in memory, i. e., the amount of assigned memory is greater than the size

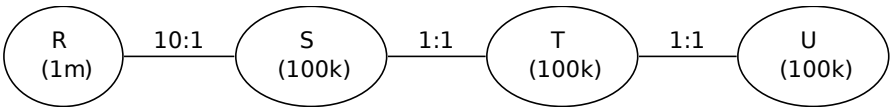


Figure 5.3: Join graph of the chain join query

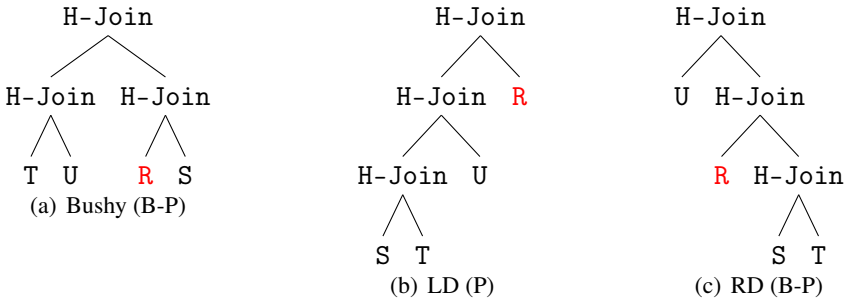
of the build input. If less memory is available, some parts (buckets) of the hash table need to be spilled to disk during build phase. In the probe phase, all tuples from the probe side that fall into spilled buckets are written to disk as well. After the probe input was fully consumed, spilled hash table buckets are loaded into memory and probed with corresponding spilled records from the probe input. In the worst-case spilling and loading must be applied recursively. The cost of a `HybridHashJoin` depends largely on the amount of memory and the size of the build input. If the build input does not fit into memory, also the size of the probe input matters because it is partially spilled to disk as well.

Sorting Query optimizers inject `Sort` operators into query plans to produce sorted output and to enable or improve the performance of subsequent operators, such as `StreamingAggregation`, `MergeJoin`, or `IndexNestedLoopJoin`. `Sort` operators require a memory budget to buffer their input. If the input completely fits into memory, a `Sort` operator only causes CPU costs. If the input does not fit into memory, an external merge-sort needs to be performed. The input is divided into multiple sorted runs each of the size of the memory budget. The runs are sequentially produced and written to disk. After the whole input was read and all runs have been produced, the sorted output is generated by reading the sorted runs from disk and merging them together. Consequently, the execution time of a sort operator depends on the size of the input data and the amount of memory. `Sort` operators deserve special attention when reasoning about plan risk. They are occasionally injected into execution plans to sort (usually) small intermediate results to improve the performance of subsequent operators, such as `IndexNestedLoopJoin` operators. Such an optional sort operator does not contribute the result computation of a query but might add an additional risk if the size of its input was underestimated.

5.1.2 Impact of Operator Order on Plan Performance

The order of operators in a query execution plan can significantly influence the size of intermediate results and the amount of data that is processed. Consequently it can also heavily affect the performance and the riskiness of a plan.

In order to evaluate the influence of operator order on the riskiness of a plan, we benchmark plans with different join orders for a query that joins four relations, R, S, T, and U. Figure 5.3 shows the join graph of the query. All relations consist of 1 KB-tuples. R contains 1 million tuples (1 GB) and S, T, and U consist of 100,000 tuples (100 MB) each. The three join predicates are $R \bowtie_{R.x=S.y} S$, $S \bowtie_{S.x=T.y} T$, and $T \bowtie_{T.x=U.y} U$, where the first join is a `10:1` join and the



(a) Bushy (B-P) (b) LD (P) (c) RD (B-P)

Figure 5.4: Join plans (left build, right probe input)

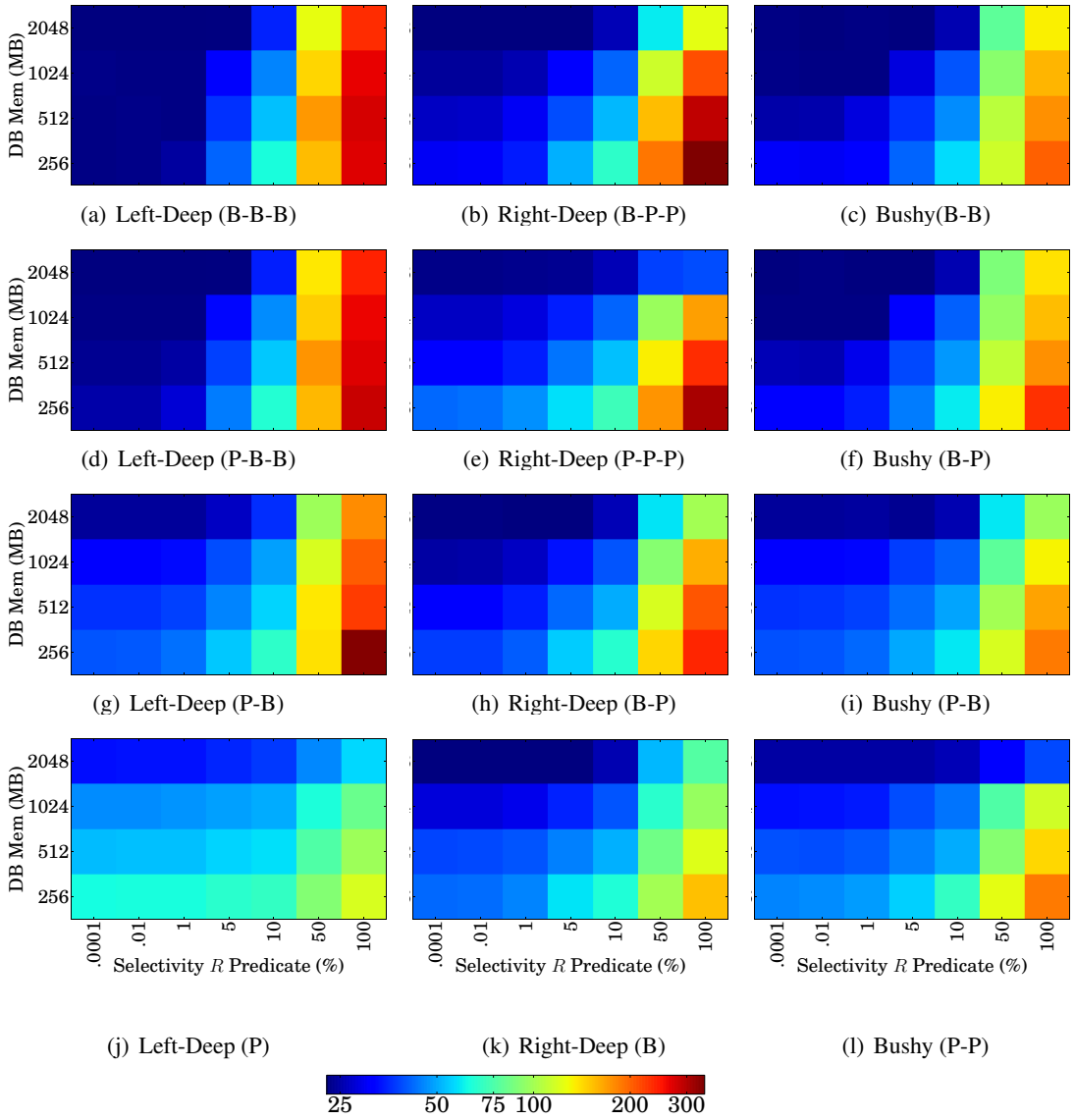
remaining two joins are 1 : 1 joins. Since S, T, and U have the same cardinality and join `1:1`, the intermediate result of joining these relations has the same cardinality but wider tuples. In addition to the three join predicates, the query has one local range predicate on R to vary the input size of the query. The selectivity of R's local predicate determines the cardinality of all intermediate results to which R contributes. Since R's cardinality is ten times as large as the cardinality of the other relations, a selectivity of less than 10% reduces and a selectivity of more than 10% increases the cardinality of intermediate results.

We constructed twelve plans to execute the query described above. All joins are executed by `HybridHashJoin` operators (left input is build, right is probe input). We place relation R on all four leaf positions of a bushy (BH), a left-deep (LD), and a right-deep (RD) join plan. Relations S, T, and U are identical and are joined in this order to prevent Cartesian products. Figure 5.4 shows a bushy, a left-deep, and a right-deep plan. "Bushy (B-P)" refers to the bushy plan where R is on the build side (B) of the first and on the probe side (P) of the second join. "Left-Deep (P)" marks the left-deep plan where R is on the probe side of the third join. Relations S, T, and U are accessed by `TableScan` operators. R is read using a `ClusteredIndexSeek` operator on R's filter attribute to avoid the fixed cost overhead of a `TableScan`.

We execute each plan for varying selectivities of R's local predicate (from .0001% to 100%) and varying memory budgets for the database system (from 256 to 2048 MB). Figure 5.5 shows the average execution time for each plan and execution condition.

The left-deep plans feed the result of a `HybridHashJoin` operator into the build side of a subsequent `HybridHashJoin` operator. Therefore, these plans perform well if the selectivity of R's predicate is low and R and S are the first relations which are joined (Figures 5.5(a) and 5.5(d)). In this case, the small result of the first join is fed into the build side of the remaining two joins. However, the same plan performs much worse if the predicate selectivity is high because `HybridHashJoin` operators are sensible to large build inputs as discussed in Section 5.1.1.

Right-deep plans forward the result of a `HybridHashJoin` into the probe input of the following `HybridHashJoin` which means that intermediate join results are processed in a pipelined fash-



(a) Left-Deep (B-B-B) (b) Right-Deep (B-P-P) (c) Bushy(B-B)

(d) Left-Deep (P-B-B) (e) Right-Deep (P-P-P) (f) Bushy (B-P)

(g) Left-Deep (P-B) (h) Right-Deep (B-P) (i) Bushy (P-B)

(j) Left-Deep (P) (k) Right-Deep (B) (l) Bushy (P-P)

Figure 5.5: Execution time (s) of different plans for a chain join query and varying input cardinality and database memory in seconds.

ion given that the hash table fits completely into memory. Therefore, these plans do not benefit from low selectivities as much as left-deep plans but also do not suffer from high selectivities.

In left- or right-deep plans at least one input of each join is a base table access. Plans with at least one join that has only intermediate results as input are called bushy plans. The third column of Figure 5.5 shows the execution times of the four bushy plans. These four plans behave similarly for varying selectivity and memory budgets. Since all relations participate in exactly two joins in each bushy plan, the benefit of small intermediate results in case of low selectivities or the penalty of large intermediate results in case of high selectivities is similar for all plans. The only performance differences of plans are due to different assignments of build and probe side.

Figure 5.5 illustrates the importance of operator order when assessing the risk of a plan. Figures 5.5(j) and 5.5(k) show the results of the left- and right-deep plan where R is joined as the last relation. Since R is the only relation whose cardinality is varied, the costs of all previous joins are constant (for constant memory). Only the execution time of the last join varies, which results in a rather low variation of the overall plan costs. Of course, joining an intermediate result with variable size late, limits the performance benefits if the result is small. In contrast, Figures 5.5(a), 5.5(d), 5.5(b), and 5.5(e) show the effect of including R in the first join of a plan.

Another observation is that plans which pipeline intermediate results of variable size (5.5(j), 5.5(e), and 5.5(l)) have quite uniform performance given that enough memory is available. Since all joins in a pipeline require their memory at the same time, his requirement might not be met in case of scarce memory resources such that the performance of pipelined plans can significantly drop (see Figure 5.5(e) for example). Note that the probe input needs to be partially written to disk if a `HybridHashJoin` operator is short on memory. In contrast, left-deep plans which do not pipeline results are less sensitive for low memory budgets because only two joins are active and consume memory at the same time.

Figure 5.5 also illustrates the common trade-off between the ability of a plan to exploit small intermediate results and the ability to gracefully handle significantly larger intermediate results. Plans that perform well in worst-case often pay with high execution overhead if less data needs to be processed.

The previous experiments illustrated a few ways in which the order of operators can affect the risk of a plan. For example plans that need to materialize intermediate results of varying size have a more varying performance than plans which are able to process intermediate results of varying size in a pipelined fashion, i. e., without materialization. Reasons to materialize intermediate results are the plan structure (early joining of varying results) and lack of memory (spilling of hash table). In fact, pipelined plans can provide low performance with little variation. However, this observation does not hold for pipelined data processing in general. For example,

Query	Involved Relations	#Plans
Q2	PS, PS, P, S, S, N, N, R, R	38
Q3	L, O, C	18
Q4	L, O	14
Q5	L, O, C, S, N, R	17
Q7	L, O, C, S, N, N	24
Q8	L, O, P, C, S, N, N, R	37
Q9	L, O, PS, P, S, N	47
Q10	L, O, C, N	21
Q11	PS, PS, S, S, N, N	16
Q12	L, O	8
Q14	L, P	4
Q16	PS, P, S	28
Q17	L, L, P	11
Q21	L, L, L, O, S, N	23
total		306

Table 5.1: Analyzed queries of the experimental study. Queries are adapted from the TPC-H benchmark. Relations with parameterized predicates are underlined and red.

pipelining of variable sized results along `NestedLoopJoin` or `IndexNestedLoopJoin` operators can cause significantly varying numbers of expensive subplan executions or random I/O operations.

5.1.3 Performance of Analytical Query Plans

In the previous section we discussed the performance characteristics of individual operators and different join orders. In this section, we report on an extensive experimental study to assess the risk of full query execution plans. We execute in total 306 different plans for 14 analytical queries on a commercial relational database system with varying selectivities of parameterized predicates and memory budgets. We measure the execution time of each plan for 108 different execution conditions and analyze this data to identify properties and features of plans which cause significantly varying plan performance. In the following, we present the experimental setup and discuss our findings in detail.

Experimental Setup The experiments are carried out using a TPC-H data set [200] of scale factor 1 and a set of modified TPC-H queries, which are taken from the Picasso project [174, 179]. The base relations are stored as clustered indexes indexed and sorted by their primary key. Each query has two parameterized local range predicates for which we vary the literal

bindings to control the predicates' selectivities. We generate a set of distinct execution plans for each query similar to the approach of Haritsa et al., i. e., we call the optimizer with different literal bindings and different physical configurations (with and without secondary indexes on the predicate attributes) and store each unique returned plan. Hence, all returned plans are considered to be optimal for at least one execution condition. Due to the varying complexity of the queries and the fact that the optimizer always returns the "best" plan, the number of plans for each query differs. See Table 5.1 for details on the selected queries and the number of generated plans.

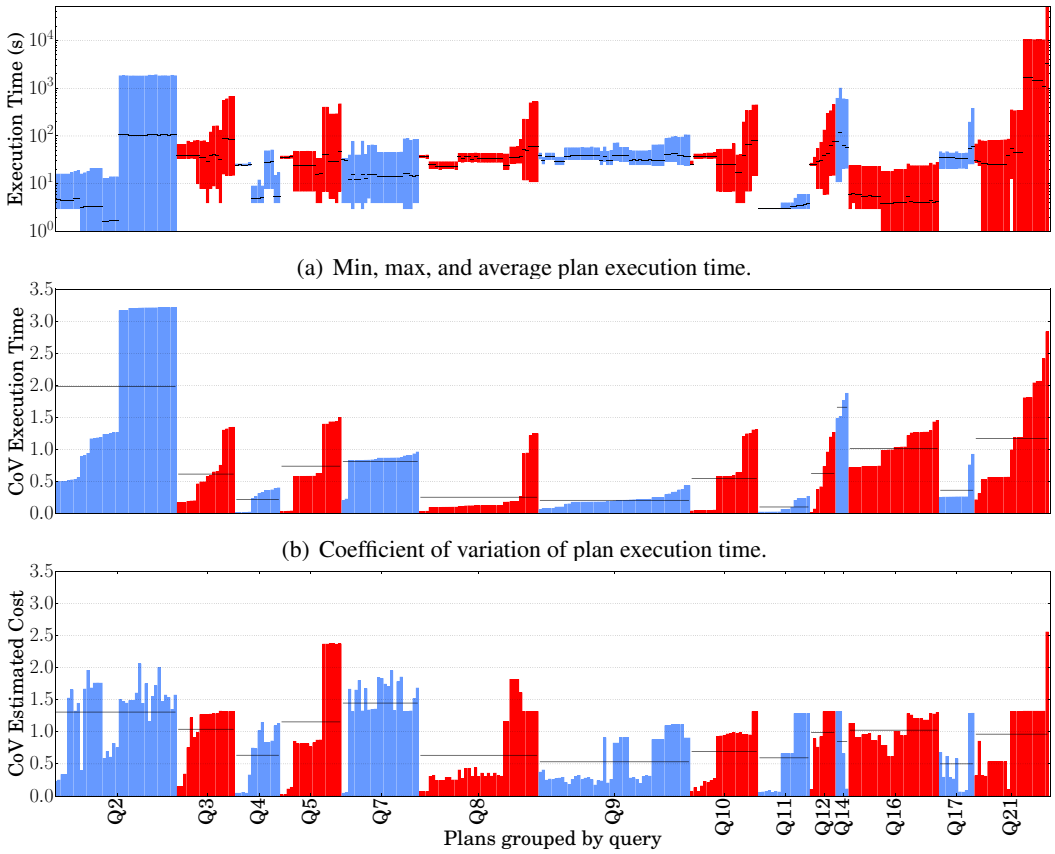
Usually a database system executes the "best" execution plan that is determined by the optimizer for a given query and execution condition. Since we want to execute each plan for every execution condition, we need to "force" the database system to execute a particular plan. We achieve this by specifying the whole plan as an optimizer hint, which leaves the optimizer almost no freedom[1]. Each plan run is carried out in isolation with an internal degree of parallelism of 1.

We run our benchmark on two standalone workstations with identical hardware and software configurations. All plan runs for a given query are performed on the same machine. Each machine is equipped with an Intel Core i7-860 CPU with 4 cores and 8 threads at 2.80 GHz, 16 GB main memory, and two 1TB SATA hard disks at 7200 rpms and with 32MB cache. One disk holds the operating system, the DBMS, and the database. The other disk is dedicated to the DMBS's temporary database, which is used to spill intermediate results to disk. The machines run a 64 Bit Windows 7 Professional with Service Pack 1. The benchmarks are executed on a current version of a commercial relational DBMS. The database system does not allow fine-grained memory configuration. Instead, we grant the DBMS a total memory budget and the DBMS internally distributes the memory among the buffer pool, workset memory, query cache, and other components.

We execute each plan with six different selectivity settings for each predicate (0.1%, 1%, 5%, 10%, 50%, and 100% selectivity) and three different database memory configurations (512MB, 1024, and 2048MB) totaling in 108 different execution conditions per plan. We compute the execution time of a plan as the average from seven execution runs, excluding the minimum and maximum observed execution time. The buffer pool of the DBMS is cleared after each run. Overall, we execute 231,336 plan runs with a total sequential execution time of about 207 days.

We continue to discuss the result of our experimental study. First, we analyze the overall variation of execution time for the executed plans. Subsequently, we have a detailed look at individual plans to identify plan features which have a significant effect on performance variations.

[1] It still chooses the assignment of memory to queries and operators.



(a) Min, max, and average plan execution time.

(b) Coefficient of variation of plan execution time.

(c) Coefficient of variation of estimated plan cost.

Figure 5.6: Actual plan execution time, coefficient of variation (CoV) of actual plan execution time and estimated plan cost grouped by query.

Overall Variation of Plan Performance Figure 5.6(a) shows the minimum, maximum, and average execution time of all 306 plans in seconds on a log scale. We see that the performance of some plans varies by multiple orders of magnitude for changing execution conditions. Even though the queried data set was only 1 GB in size, some plans for query Q21 run for more than two hours in worst case but take only subseconds for low predicate selectivities. However, we also find plans with low performance variation.

Figure 5.6(b) shows for each plan the coefficient of variation (CoV) or relative standard deviation of the observed execution times for all execution conditions. The coefficient of variation c_v is defined as the ratio of the standard deviation σ and the average μ, $c_v = \frac{\sigma}{\mu}$. The plans are shown in the same order as in Figure 5.6(a) (grouped by query, in ascending order of CoV). The black lines indicate the average CoV of each query. Over all plans, the CoV ranges from 0.017 (Q4) to 3.213 (Q2). Within a single query, the CoV can also differ significantly as for

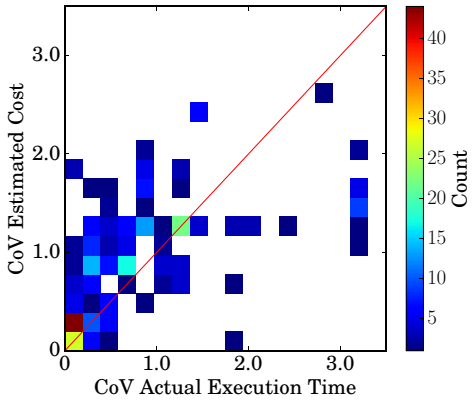


Figure 5.7: CoV of actual execution time vs. CoV of optimizer estimated cost.

example from 0.218 to 3.119 for plans of query Q21. In contrast, the CoV values for plans of query Q11 range only from 0.027 to 0.272. When comparing Figures 5.6(a) and 5.6(b), a few intuitions can be confirmed. Plans with low performance variation have often constant, but in comparison to other plans of the same query a rather high, minimum execution time as shown by queries Q4, Q5, Q10, Q12. However, there are also exceptions as for example queries Q9, Q11, and Q17. Plans with high performance variations have often significantly higher maximum execution times compared to other plans of the same query. Again there are exceptions, such as some plans for queries Q4, and Q16. For some queries (e. g., Q2, Q4, Q5, and Q8), we can identify plans with lower average and similar or lower maximum execution time compared to the plan with the least performance variation. These plans perform better on average and similar (or better) in the worst case but have a higher variation in execution time.

The comparison of the minimum, the maximum, and the CoV of the plans' execution times shows that the choice of the optimal plan for a query is not trivial. For most of the plans analyzed, low performance variation correlates with rather high minimum execution time. Hence, there is a trade-off between robust and optimistic plan choices, which depends on the concrete use case. While in some situations robust plan choices are favorable, optimistic plan choices are better suited for other situations. However, we also see that there are a few plans for some queries, which combine the best of both worlds, i. e., a low minimum execution time and low performance variation. In any case, the risk of a plan needs to be assessed in order to be able to take this information into account when choosing an execution plan. As previously noted, each plan shown in Figure 5.6 is chosen by the optimizer and considered to be optimal for some execution condition. Consequently, these plans are a biased sample from the queries' whole plan spaces.

Figure 5.6(c) shows the coefficient of variation of all plans' optimizer cost estimates for all execution conditions. Plans are in the same order as in Figure 5.6(a) and Figure 5.6(b). Comparing Figures 5.6(b) and 5.6(c), we see for several plans a clear difference of the CoV of estimated

plan costs and the CoV computed from actual execution time. For queries Q5, and Q10, the execution time CoV is quite well captured by the optimizer's estimates. However for queries Q2, Q3, Q9, and Q17 the estimated cost CoV of some plans differ significantly from their actual performance variation. In general, the CoV computed from the optimizer's estimates exhibit less variation compared to the CoV computed from actual plan execution time. Looking at the average execution time CoV by query, the minimum and maximum CoV of execution time are 0.106 and 1.987, while the minimum and maximum CoV of estimated costs are 0.501 and 1.448.

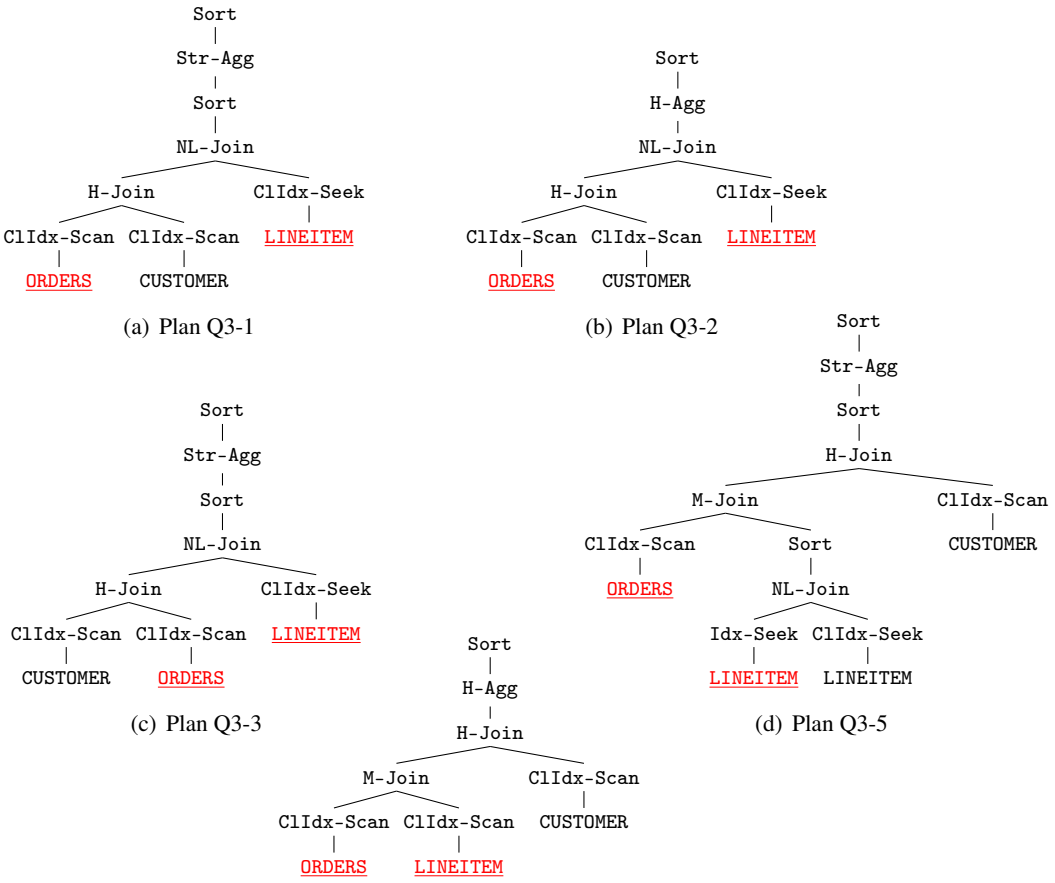
Figure 5.7 shows a binned scatterplot of the CoV of actual execution time and estimated cost for all benchmarked plans. The red line indicates exactly matching CoVs. The plot visualizes that assessing the performance variation of a plan using optimizer estimates does not yield accurate results in general. In general, the estimated cost CoV for plans with low actual risk is quite precise. However, there are also exceptions. For example, the estimated cost CoV for plans with an actual execution time CoV between 0.0 and 0.2 ranges from 0.0 to 2.0.

Identifying Risky Plan Features After we had a look at the variance of performance of our benchmarked plans, how it relates to their execution time, and how the CoV of actual execution time and estimated cost differ, we continue to analyze the execution time of individual plans for changing execution conditions. We identify plan features, which are responsible for varying performance, and verify some of the observations made in previous sections with the help of concrete example plans.

```
SELECT
    l_orderkey, o_orderdate, o_shippriority,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer, orders, lineitem
WHERE
    c_mktsegment = 'BUILDING'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_totalprice < ?
    AND l_extendedprice < ?
GROUP BY
    l_orderkey, o_orderdate, o_shippriority
ORDER BY
    revenue desc, o_orderdate
```

Figure 5.8: Query Q3

We start by looking at some plans for query Q3. The SQL statement of the query is shown in Figure 5.8. It joins the `customer`, `orders`, and `lineitem` relations, computes an aggregate, and sorts the result. All relations are restricted by local predicates. While the predicate on

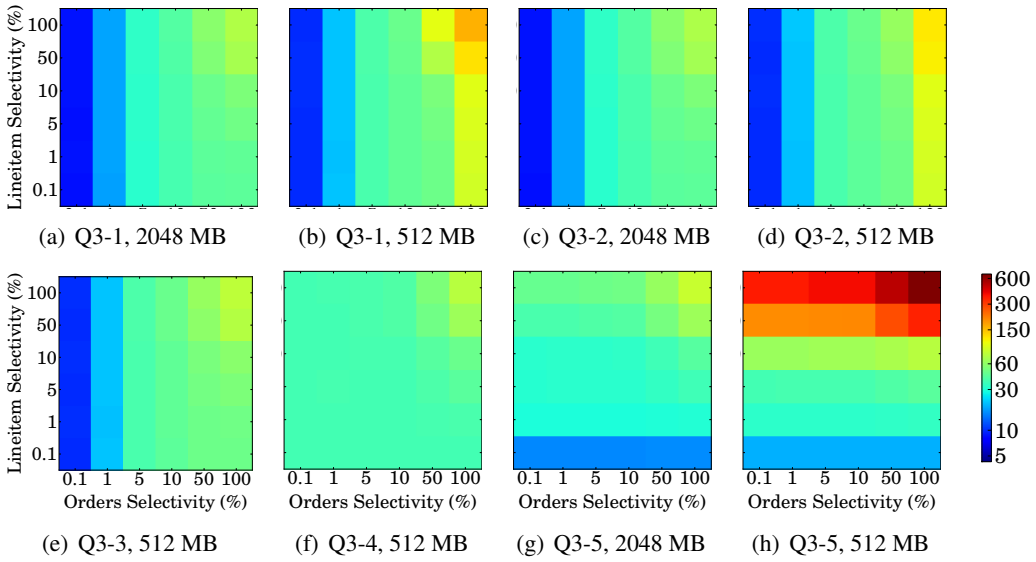


(a) Plan Q3-1 (b) Plan Q3-2 (c) Plan Q3-3 (d) Plan Q3-5 (e) Plan Q3-4

Figure 5.9: Query Q3 execution plans

`customer` is fixed, the predicates on `orders` and `lineitem` are parameterized to vary their selectivity. Figure 5.9 shows five plans for query Q3, which we analyze in detail. Relations with parameterized predicates are underlined and shown in red color. We compare plans that only differ in specific details to isolate the effect of certain plan features. That way we are able to reason about the effect of individual plan features on the performance variance of a plan.

Figure 5.10 shows the execution time of the five selected plans for different memory settings and varying selectivities of both parameterized predicates. Plan Q3-1 uses a `HybridHashJoin` to join `orders` and `customer` with `orders` being the join's build input. The result is joined with the `lineitem` relation using an `IndexNestedLoopJoin`, and then sorted, stream-aggregated, and sorted again. Figure 5.10(a) shows the execution time of plan Q3-1 for 2048 MB database memory. The performance of the plan mostly depends on the selectivity of the `orders` predicate



(a) Q3-1, 2048 MB (b) Q3-1, 512 MB (c) Q3-2, 2048 MB (d) Q3-2, 512 MB

(e) Q3-3, 512 MB (f) Q3-4, 512 MB (g) Q3-5, 2048 MB (h) Q3-5, 512 MB

Figure 5.10: Query Q3 Plan execution times (s) for varying predicate selectivities and database memory.

and ranges from 8 to 74 seconds. Figure 5.10(b) shows the execution time of the same plan but with database memory reduced to 512 MB. For lower selectivities the performance is the same as with 2048 MB. However, for selectivities of 50% and more, the execution time increases to a maximum of 162 seconds, a factor of two. This regression of performance for high predicate selectivities and low memory is due to the memory-sensitive `HybridHashJoin` and `Sort` operators, which spill data to disk if their (build) inputs do not fit into memory and the increased number of random I/O operations of the `IndexNestedLoopJoin` due to the reduced size of the buffer pool.

Plan Q3-2 (Figure 5.9(b)) differs from plan Q3-1 in how it computes the aggregation. While plan Q3-1 uses a `StreamAggregate` operator with an additional `Sort` operator, plan Q3-2 employs a `HashAggregate` operator, which does not require sorted input. Figures 5.10(c) and 5.10(d) show the execution time of plan Q3-2 for 2048 and 512 MB database memory respectively. While Q3-1 and Q3-2 have similar performance for 2048 MB database memory, the worst-case execution time of Q3-2 is lower than Q3-1 ones (120 vs. 162 seconds). This difference is caused by the additional `Sort` operator of Q3-1, which is sensitive to increasing data sizes and decreasing memory budgets as discussed in Section 5.1.1.

Both plans, Q3-1 and Q3-2, have another plan feature that causes increasing execution time for decreasing database memory. Comparing plan Q3-3 (Figure 5.9(c)) with Q3-1 we see that the only difference are the switched input sides of the `HybridHashJoin` operator, i. e., `customer`

becomes build side and orders becomes probe side. This change improves Q3-3's robustness in two ways compared to plan Q3-1. First, the build side of the HybridHashJoin operator has a fixed size (the selectivity of the predicate on customer is not varied) such that the memory requirement of the hash table remains constant. Second, the input of the subsequent IndexNestedLoopJoin becomes sorted on lineitem's join attribute. This is because, the orders relation is sorted on its primary key due to the clustered index and the sorting is preserved because orders is the probe side of the HybridHashJoin. Since lineitem is also stored in a clustered index organized by the join attribute, the IndexNestedLoopJoin accesses the disk pages of lineitem's clustered index in a sequential pattern. Due to the buffer pool, only the first read of each page goes to disk. All following reads are logical reads and served from the buffer pool. Since the outer side of the IndexNestedLoopJoin is not sorted in plans Q3-1 and Q3-2, the index accesses are randomly distributed over the whole index. In this case, the probability of a buffer pool hit increases with increasing buffer pool size. Therefore, plans Q3-1 and Q3-2 are more sensible to reductions of database memory as shown by the execution time plots for 512 MB database memory of the three plans (Figures 5.10(b), 5.10(d), and 5.10(e)). For 512 MB, plan Q3-3's execution time ranges from 10 to 80 seconds (8 to 74 seconds for 2048 MB).

Plan Q3-4 (Figure 5.9(e)) joins orders and lineitem using a MergeJoin. Since both relations are stored in a clustered index sorted on their join attribute, no additional Sort operator is required for this join. The result of the join is fed into the build side of a HybridHashJoin. The remaining customer relation is the probe input. The result is aggregated using a HashAggregation operator and finally sorted. Compared to the other plans for Query 3, Plan Q3-4's execution time is rather constant (38 to 78 seconds for 512 MB database memory) because it performs full sequential scans of all three relations (no reduced I/O for lower predicate selectivities), uses a MergeJoin operator with robust execution behavior, and a hash aggregation. The only possible source of major performance variation is the variable sized build input of the HybridHashJoin operator. However, since this join is the only major memory consumer in the plan, its impact on the overall execution time is marginal. The performance of plan Q3-4 is also constant for changing database memory settings. Its maximum execution time only increases from 65 to 78 seconds when reducing the memory from 2048 MB to 512 MB.

Plan Q3-5 (Figure 5.9(d)) differs from plan Q3-4 in the way the lineitem relation is accessed. It uses a secondary index on the attribute of the variable local predicate, joins it with the clustered index to construct the required tuple and sorts the result for the subsequent MergeJoin with the orders relation. In addition, the aggregation is performed using a Sort and a StreamAggregation operator. Figures 5.10(f) and 5.10(h) show the execution time of Q3-4 and Q3-5 for 512 MB database memory, respectively. While the execution time of plan Q3-4 differs by slightly more than a factor of two (38 to 78 seconds), the execution time of plan Q3-5 varies significantly from 19 (low selectivity of lineitem predicate) to 670 seconds (high selectivity of both predicates), a factor of 35. The main reasons for the significant performance difference with 512 MB memory are the secondary and clustered index seeks and the following Sort required

by the MergeJoin operator. These three operators are very sensitive to low memory budgets, the index seeks due to the size of the buffer pool and the sort due to the size of its workset memory, which might force it to spill to disk in case of insufficient memory. Looking at the performance of plan Q3-5 with 2048 MB of database memory (Figure 5.10(g)) we see that the maximum execution time is reduced to 86 seconds.

```
SELECT
    n_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue
FROM
    customer, orders, lineitem, supplier, nation, region
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND l_suppkey = s_suppkey
    AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'ASIA'
    AND o_orderdate >= '1994-01-01'
    AND o_orderdate < '1995-01-01'
    AND c_acctbal < ?
    AND s_acctbal < ?
GROUP BY
    n_name
ORDER BY
    revenue DESC
```

Figure 5.11: Query Q5

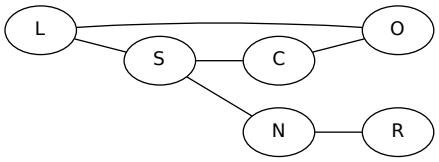


Figure 5.12: Query Q5 join graph

Another plan feature with potentially large impact on plan risk is illustrated by three plans for query Q5. Query Q5 is shown in Figure 5.11 and joins six tables, customer, orders, lineitem, supplier, nation, and region. The relations region and orders are filtered with constant local predicates and the two parameterized predicates are applied on the customer and supplier relations. An interesting aspect of this query is the cyclic join graph (see Figure 5.12) with the join between customer and supplier on their nationkey attributes being not a primary-key/foreign-key join. The result of joining all relations is grouped, aggregated, and sorted.

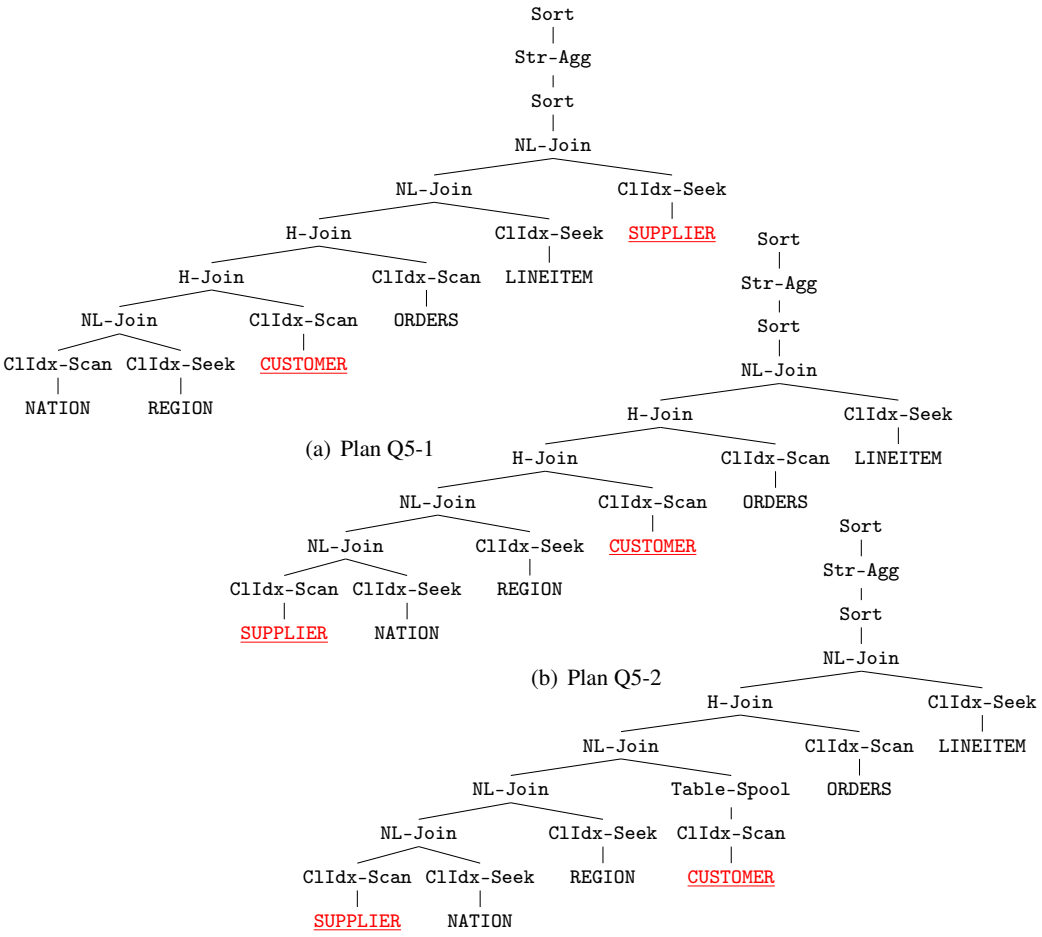
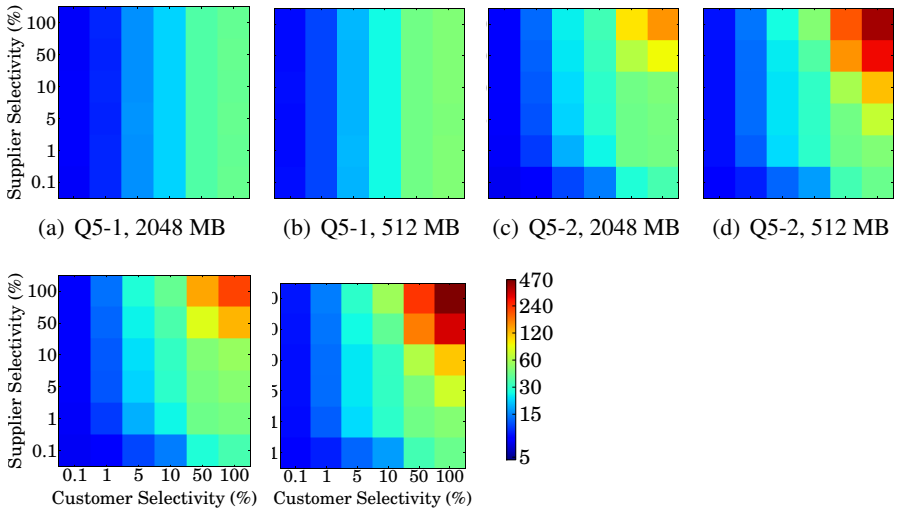


Figure 5.13: Query Q5 execution plans

Plan Q5-1 is shown in Figure 5.13(a) and is a left-deep plan consisting of `IndexNestedLoopJoin` and `HybridHashJoin` operators followed by a `Sort`, `StreamAggregate`, and final `Sort` operator. Figures 5.14(a) and 5.14(b) show the execution time of plan Q5-1 for 2048 and 512 MB database memory, respectively. The execution time of the plan is determined by the selectivity of the parameterized predicate on `customer` because it determines the size of the intermediate results that are given to the build input of the `HybridHashJoin` with the `orders` relation and the outer sides of the `IndexNestedLoopJoin` operators with the `lineitem` and `supplier` relations. The minimum and maximum execution time of plan Q5-1 for 512 MB memory are 8 and 47 seconds, respectively (7 and 41 for 2048 MB memory).

Similar to plan Q5-1, plan Q5-2 (Figure 5.13(b)) is also a left-deep join plan with a `Sort`, `StreamAggregate`, and final `Sort` operator on top as in plan Q5-1. While most relations are



(a) Q5-1, 2048 MB (b) Q5-1, 512 MB (c) Q5-2, 2048 MB (d) Q5-2, 512 MB

(e) Q5-3, 2048 MB (f) Q5-3, 512 MB

Figure 5.14: Query Q5 plan execution time (s) for varying predicate selectivities and database memory.

joined using the same join operators, the order in which they are joined is different. The execution time of plan Q5-2 for 512 MB database memory is shown in Figure 5.14(d) and ranges from 8 seconds to 401 seconds. Figure 5.14(c) shows the performance for 2048 MB memory (7 to 148 seconds). The significant difference in maximum execution time compared to plan Q5-1 is caused by the non-primary-key/foreign-key m:n join of the `supplier` and `customer` relations. Since there are only 25 distinct values of the `nationkey` join attribute, the intermediate result of the join is very large. The large intermediate result is fed into the build input of the `HybridHashJoin` with `orders` and even more importantly into the outer input of the `IndexNestedLoopJoin` with the `lineitem` relation. The join with `lineitem` finally closes the cyclic join graph and significantly reduces the size of the intermediate result. The strong dependency of plan Q5-2's performance on the amount of memory can be accounted to the smaller buffer pool to cache pages of the `lineitem` relation and smaller memory budget for the hash table of the `HybridHashJoin` with `orders`.

Plan Q5-3 is shown in Figure 5.13(c) and is identical with plan Q5-2 except in the way the `customer` relation is joined. In plan Q5-2, the `customer` relation is read by a `ClusteredIndexScan` and joined as the probe side of a `HybridHashJoin`. In plan Q5-3, the `customer` relation is also read with a `ClusteredIndexScan`, but the filtered and projected result (only the `custkey` and `nationkey` are required) is written into a temporary table. The temporary table serves the inner input of a `NestedLoopJoin`. Consequently, the temporary table is repeatedly scanned by the `NestedLoopJoin`. How often this happens depends on the size of the outer input. Figures 5.14(e) and 5.14(f) show the performance of plan Q5-3 for varying predicate selectivities and 2048 and 512 MB database memory, respectively. Same as plan Q5-2, plan Q5-3 suffers

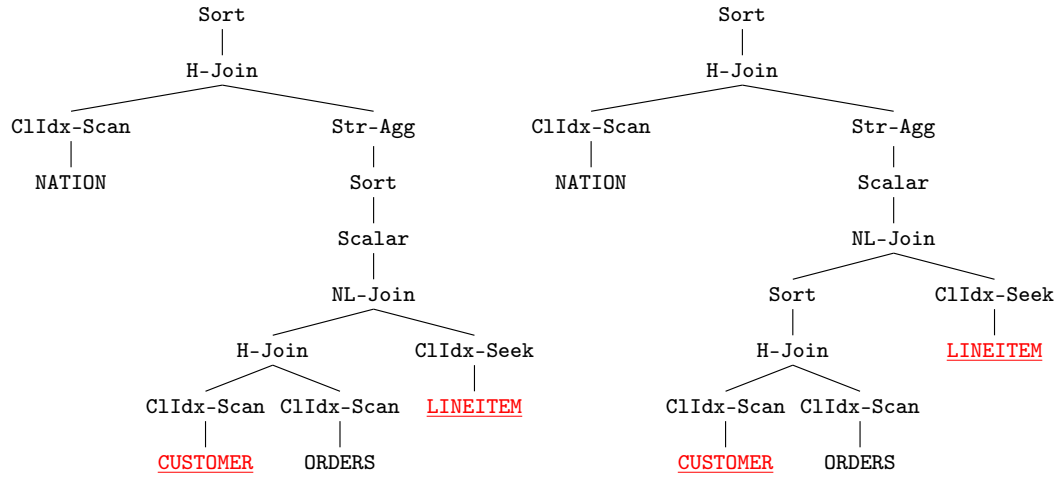
from the large intermediate result caused by the m:n join between customer and supplier. Therefore, the patterns of their execution time plots are quite similar, although plan Q5-3 is even worse than plan Q5-2. While plan Q5-2's performance ranges from 7 to 148 seconds for 2048 MB memory (8 to 401 seconds for 512 MB memory), Q5-3 takes 7 to 220 seconds to executed with 2048 MB memory and 8 to 470 seconds with 512 MB memory. Compared to Q5-2 the maximum execution time of plan Q5-3 is about 70 seconds higher independent of the amount of memory. This example shows the impact that a NestedLoopJoin operator with an outer side of uncertain size can have on the risk of a plan.

```
SELECT
    c_custkey, c_name,
    SUM(l_extendedprice * (1 - l_discount)) AS revenue,
    c_acctbal, n_name, c_address, c_phone, c_comment
FROM
    customer, orders, lineitem, nation
WHERE
    c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate >= '1993-10-01'
    AND o_orderdate < '1994-01-01'
    AND c_nationkey = n_nationkey
    AND c_acctbal < ?
    AND l_extendedprice < ?
GROUP BY
    c_custkey, c_name, c_acctbal, c_phone,
    n_name, c_address, c_comment
ORDER BY
    revenue DESC
```

Figure 5.15: Query Q10

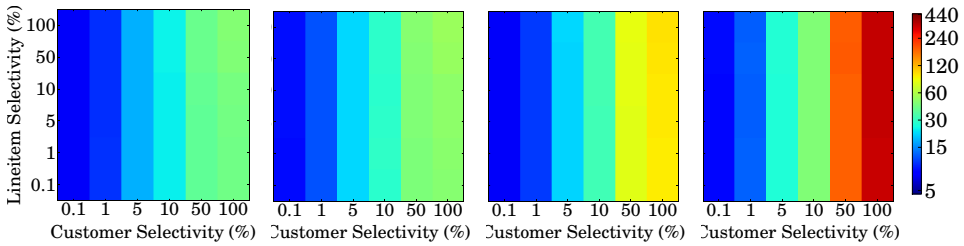
Two plans from query Q10 illustrate how opportunistic optimizer decisions can lead to extreme performance variations without significant execution time improvements. Query Q10 joins four relations, lineitem, orders, customer, and nation, performs an aggregation, and sorts the result (see Figure 5.15. Fixed local predicates are applied on orders and parameterized predicates are applied on lineitem and customer.

Figure 5.16 shows two execution plans for query Q10. The plans are identical except for the position of a Sort operator. Both plans start joining the customer and orders relation using a HybridHashJoin operator with the orders relation being on the probe side. In plan Q10-1 the result of the join is forwarded into the outer side of an IndexNestedLoopJoin with lineitem. The result is sorted on custkey and stream-aggregated. In plan Q10-2, the result of the first join is first sorted on custkey, then used as the outer input of the IndexNestedLoopJoin with lineitem, and after that stream-aggregated. Both plans continue with joining their result with



(a) Plan Q10-1 (b) Plan Q10-2

Figure 5.16: Query Q10 execution plans



(a) Q10-1, 2048 MB (b) Q10-1, 512 MB (c) Q10-2, 2048 MB (d) Q10-2, 512 MB

Figure 5.17: Query Q10 plan execution time (s) for varying predicate selectivities and database memory.

the `nation` relation and finally sorting the result.

The `Sort` operator is required by the `StreamAggregate` operator. The optimizer has the option to move the `Sort` operator below the `IndexNestedLoopJoin` operator because it preserves the order of its outer input. In case of plan Q10-2, the optimizer decides to move the sort below the join because this reduces the amount of data to be sorted due to the smaller tuple width.

Figure 5.17 shows the execution time of both plans for 2048 and 512 MB database memory. For both plans, the performance depends on the selectivity of the `customer` predicate. The performance of plan Q10-2 also varies for changing memory settings. While the execution time of plan Q10-1 ranges from 7 to 47 seconds, the execution time of plan Q10-2 ranges from 7 to 98 seconds for 2048 MB database memory (8 to 53 and 8 to 340 seconds for 512 MB). This significant difference in performance is caused by the different position of the `Sort` operator in both plans, which are identical otherwise.

The position of Sort operator significantly influences the performance of the IndexNestedLoopJoin with the lineitem relation. In plan Q10-1 the outer input of the join is sorted on the orderkey because the probe side of the previous HybridHashJoin is a scan of the clustered index on orders, which is sorted on orderkey, its primary key. Because the clustered index on lineitem is also sorted on orderkey and orderkey is the join attribute between orders and lineitem, the IndexNestedLoopJoin in plan Q10-1 sequentially reads the disk pages of the clustered index on lineitem. Plan Q10-2 sorts the output of the first join on custkey before it is fed into the outer side of the IndexNestedLoopJoin with lineitem. Therefore, the ClusteredIndexSeek operator causes many expensive random I/O accesses. Again, a large buffer pool can reduce the number of I/O operations that actually hit the disk.

5.1.4 Identified Risky Plan Features

In the previous sections we discussed the impact of individual operators, join order, and pipelined execution on the execution time of query plans and analyzed complete execution plans and their performance for changing input sizes and memory settings. We identified several plan features, which can be responsible for a significantly varying performance of execution plans and hence can have a negative impact on the robustness of an execution plan. In the following we give a summary of these features.

NestedLoopJoin The performance of a NestedLoopJoin heavily depends on the cardinality of its outer input. Since it evaluates its inner side for every outer tuple (or batch of outer tuples), it becomes expensive even for small numbers of tuples on the outer side. To reduce this cost, optimizers sometimes inject a temporary table to cache the result of the inner side (see Figure 5.13(c) for example). However even in such cases, the cost of a NestedLoopJoin can be high if the size of the outer or inner side is underestimated. Due to its high costs even for small cardinalities, a NestedLoopJoin is seldom used for equality joins where usually better alternatives exist.

IndexNestedLoopJoin The IndexNestedLoopJoin is a "common special case" of the NestedLoopJoin and can be a good operator to join a large indexed relation. Its execution time does also depend on the cardinality of its outer input because the join performs an index seek operation for each outer tuple. In principle, each index seek operation causes at least one expensive random I/O operation. The cost the index seeks can be reduced if the target relation (partly) fits into the buffer pool. If the tuples of the outer input are sorted on the join attribute and the accessed index is clustered, the random I/O pattern becomes sequential and disk pages might even be asynchronously pre-fetched. In this case, the cost variance of the IndexNestedLoopJoin decreases significantly. Plans Q10-1 and Q10-2 in Section 5.1.3 illustrate the effect of sorted outer sides.

Index Seeks `IndexSeek` operators are used to apply a predicate on a relation. The predicate can be a local predicate or a join predicate if the `IndexSeek` operator is called from an `IndexNestedLoopJoin`. In the latter case, the `IndexSeek` operator is called several times with different value bindings of the join attribute. If the seek operation is performed on a clustered index, the number of random I/O operations is independent of the predicate's selectivity, because the data is physically organized in order of the predicate attribute. In case of a `SecondaryIndexSeek` operator, each tuple that fulfills the predicate condition might be located on a different disk page and cause an additional random I/O operation. Hence, `SecondaryIndexSeek` operations are very sensitive for changing predicate selectivities and can result in significant execution costs. A large buffer pool helps to reduce the amount of physical I/O operations. Figure 5.2(c) and Plan Q3-5 (Figures 5.9(d), 5.10(h), and 5.10(g)) illustrate this behavior.

M:N Joins The result cardinality of an equi-join where at least one join attribute has a uniqueness constraint (such as `UNIQUE` or `PRIMARY KEY`) is bounded by the cardinality of the relation with the non-unique join attribute. In contrast, non-equality joins or equality-joins on non-unique attributes can result in large intermediate results. Moreover, the result cardinalities of non-equality and equality joins on non-unique attributes are hard to estimate such that optimizers can be easily off by multiple orders of magnitude. Hence, such joins pose a high risk and should be avoided if possible. Plan Q5-2 (Figures 5.13(b), 5.14(c), and 5.14(d)) illustrates which effect such a join and its intermediate result can have on the performance of a plan.

Sort While some queries explicitly ask for ordered results, `Sort` operators are also often injected by optimizers. One reason to add a sort operator are subsequent operators that require sorted input, such as `MergeJoin` or `StreamAggregate`. Another reason to inject a sort operator is to improve the performance of an operator, as for instance an `IndexNestedLoopJoin` on a clustered index. However, most sort-based operators have also hash-based alternatives that do not need sorted input. Usually, a sort-based operator and the necessary `Sort` operator are chosen over a hash-based alternative if the optimizer expects a small input size. However, the `Sort` operator can become expensive if its input data exceeds its workspace memory such that it needs to write data to disk and read it back (see Plans Q3-1 and Q3-2 in Section 5.1.3). A few times, we observed that the optimizer injected a `Sort` operator to sort the outer side of an `IndexNestedLoopJoin`. In such cases, the optimizer expects that the additional costs of the sort are less than the costs savings of the sequential access pattern of the `IndexNestedLoopJoin`. However, Plan p10-2 shows the result of an inverse case, where the optimizer pushes a `Sort` operator in front of an `IndexNestedLoopJoin` that shuffles the index accesses and causes the plan to become expensive for high selectivities.

HybridHashJoin Finally, hash joins read their build input and create an in-memory hash table. If the memory is not sufficient, parts of the data are spilled to disk (see Plans Q3-1 and

Q3-3 in Section 5.1.3). Therefore, the performance hash joins depend on the size of their build input. However, the performance of a `HybridHashJoin` degrades rather gracefully as only parts of the build and probe input need to be spilled to disk.

Similar observations were published in a technical report by Abhirama et al. [2]. In this report, the authors propose a method to reduce the plan space of a query by replacing the optimal plan at some locations of the plan space with neighboring plans that have costs close to the optimal plan, but which exhibit a more stable behavior for other regions of the plan space. The authors made a few observations regarding common changes made by their method, i. e., how did the plans which were replaced differ from the replacing plans. We cite and comment the technical report [2] in the following:

- *"Index intersections are often replaced by joins based on sequential scans. This is due to the indexes becoming very expensive at the higher selectivity regions of the selectivity space."* We did not look at plans with intersecting indexes in our study. However, index seek operations, either for local predicate evaluation or as inner side of `IndexNestedLoopJoin` operators, are frequently responsible for significantly varying plan performance.
- *"Nested-loop-based plans are frequently replaced with hash-join-based plans, but the reverse was never observed. Further, merge joins were almost never retained."* The analysis of our benchmark showed that `NestedLoopJoin` operators are indeed risky choices if the cardinality of their outer side varies. However, there are some exceptions for `IndexNestedLoopJoin` operators. Our observations showed that `MergeJoin` operators are robust choices given that the input is already sorted (e. g. due to a `ClusteredIndexScan`). In case that an input with variable size needs to be sorted, using this input on the probe side of a `HybridHashJoin` operator is often a more robust choice.
- *"Finally, we also often saw that the join order of the replacement plan was different to that of the original plan. In particular, left-deep plans were typically replaced by bushy plans."* Our experiments in Section 5.1.2 shows that left and right deep plans can easily degrade. Figure 5.5 shows that a pipelined plan with variable probe side is hard to beat given that enough memory is available to hold all hash tables in memory. Otherwise, bushy plans offered a good compromise between good performance for small input sizes and not-too-bad performance for large input sizes.

5.2 Defining Plan Risk and Using it for Safe Plan Choices

In the previous sections we discussed why it is desirable to distinguish robust from risky plans and identified plan features that can cause significant variations in execution time. So far we used the term "robust plans" for plans that exhibit only low or moderate performance variations for changing execution conditions and called plans risky if their execution time was highly sensitive to varying execution conditions. In order to assess and compare the robustness of query execution plans we need a metric that captures the riskiness of a plan. In this section we propose such a plan risk score to compute and compare the risk of execution plans. Subsequently, we propose a method that leverages the plan risk score to compute a risk-weighted plan cost in order to prevent the choice of very risky execution plans.

5.2.1 Defining a Risk Score for Execution Plans

The goal of the plan risk score is to give a quantitative measure for the performance variation of a plan with respect to changing execution conditions. The considered execution conditions may include several varying parameters, such as changing memory budgets, predicate selectivities, or data set sizes. In fact, the choice of the variable parameters depends on the availability of information and the application of the risk score. For example the selectivity of a parameterized predicate might be considered while the selectivities of a fixed predicate is not included because it can be accurately estimated. The amount of available memory can be a varying parameter of the execution condition in setups that concurrently run ad-hoc queries.

In the following, we present our definition of the plan risk score. We represent each varying parameter of the considered execution conditions as a continuous random variable D_i and the considered execution conditions as a random vector $\vec{D}$ consisting of all n random variables $D_i, 1 \leq i \leq n$. The probability density function $f_{\vec{D}}(d)$ of the random vector $\vec{D}$ describes the probabilities of the considered execution conditions. The function $t_p(d)$ gives the execution time of a plan p for an execution condition d. Note that the execution time of the plan only depends on d because all other parameters that influence p's performance are assumed to be fixed. We define T_p as a continuous random variable that describes the execution time of a plan p with respect to the random vector $\vec{D}$ of all considered plan execution conditions as

$$T_p = t_p(\vec{D}). \tag{5.1}$$

Given T_p, we define the plan risk score r_p of plan p for the considered execution conditions as the coefficient of variation (CoV) or relative standard deviation of its execution time distribution

$$r_p = \frac{\sigma(T_p)}{\mu(T_p)}. \tag{5.2}$$

Similar to the coefficient of variation of a plan's execution time distribution, we can also compute the coefficient of variation of a plan's distribution of estimated costs.

Wiener et al. propose a benchmark for query execution robustness [204] and define the metric "consistency" as the variance of plan execution time, which is very close to our definition. While a variance-based risk score captures the magnitude of the differences in execution time, defining the plan risk score as the coefficient of variation gives the nice property that the risk score is independent of the unit used. Hence, it is possible to compare plan risk values computed from optimizer-estimated costs and from performance measurements. Further, the coefficient of variation allows to compare the risk of plans (even for different queries) with execution times that differ by orders of magnitude due to the normalization by the average execution time.

5.2.2 Using Plan Risk Scores to Compute Risk-weighted Plan Costs

Conventional cost-based query optimizers chose query execution plans based on cost estimates, which are obtained for an assumed execution condition. As discussed before, assumed execution conditions can significantly differ from actual conditions at the time when a plan is executed. A plan which performs well for the assumed execution condition might perform poorly for the actual condition. Since plans can substantially differ in their sensitivity for changing execution conditions, a plan with more uniform performance can be a better choice.

Situations where a plan behaves much worse than expected due to unknown execution conditions could be prevented if the optimizer would take the risk of a plan into account. However, ranking execution plans based on estimated execution cost and risk is not straightforward. Figure 5.6 shows that plans with low performance variation often exhibit a rather high (but constant) execution time. In contrast, plans with high variation are potentially fast but might take a long time to complete under certain conditions. In this section, we propose and evaluate a method that computes risk-weighted plan costs by combining the previously defined risk score and the estimated costs of plans.

Our method aims to approximate the cost distribution of a plan using the plan's risk score and derives the risk-weighted cost estimate as a percentile from this distribution. The percentile is a user-specified parameter q, which weighs performance and robustness of plan choices. Given a plan p and an uncertain execution condition, such as a parameterized predicate with uncertain selectivity, our method requires an optimizer cost estimate for the assumed execution condition c_p^*, a maximum cost estimate c_p^+ for the worst-case execution condition (such as selectivity = 100%), and a risk score r_p. Given this information, we approximate the cost distribution as a normal distribution with mean $\mu = c_p^*$ and standard deviation $\sigma = r_p * (c_p^+ - c_p^*)$ and choose a risk-weighted cost estimate as the q-th percentile of this distribution. The rational of this approximation is as follows. If a plan has a low risk score or the assumed execution condition is close to the worst-case execution condition, the standard deviation of the cost distribution is

small and hence the risk-weighted cost estimate is close to the estimated cost of a conventional cost-based optimizer. For plans with a high risk and an assumed execution condition which is far from the worst-case condition, the standard deviation is large and the risk-weighted cost estimate can be significantly higher than the conventional cost estimate.

In the following, we evaluate the proposed method and show that risk-weighted cost can prevent plan choices that result in exceptional high execution times. The evaluation is based on the data we collected in the course of our experimental study (Section 5.1.3) where we executed 306 query plans for 108 different execution conditions (6 different selectivities for each of two local range predicates and 3 database memory configurations). We simulate plan choices of different methods for a given query and an assumed execution condition and analyze the execution time of the chosen plans for actual execution conditions, which are different from the assumed one. For this evaluation, we limit the differences of assumed and actual execution conditions to a single predicate selectivity. This means, for each evaluation we let estimated and actual selectivities only diverge for one of the two range predicates (see Table 5.1 as a reference) and provide exact estimates for the other range predicate and the memory setting. We use linear interpolation of our benchmark data to obtain the optimizer cost estimate and the execution time of a benchmarked plan for an arbitrary execution condition.

For the evaluation we compare the execution times of a plan chosen by the risk score method (RSP) to the best possible plan choice (BP), i. e., the plan with the lowest execution time for the *actual* execution condition, and also to the plan with the minimum estimated cost for the *assumed* execution condition (MCP), i. e., the plan that a conventional cost-based optimizer would choose. We choose plans for six assumed selectivities, 0.1%, 1%, 5%, 10%, 50%, and 100%. For each assumed selectivity, we randomly pick 500 actual selectivities either from a normal distribution with the estimated selectivity as mean and 10% selectivity as standard deviation truncated to 0% and 100% (we used the stats.truncnorm distribution from SciPy 0.16.0 [185]) or from a uniform distribution between 0% and 100%. The plan risk scores used for the RSP method are computed from actual execution times for six selectivities 0.1%, 1%, 5%, 10%, 50%, and 100% that we measured in the course of our experimental study (Section 5.1.3).

Table 5.2 compares the execution times of minimum estimated cost plans (MCP) and risk-weighted cost plans (RCP) to the execution times of the best possible plan (BP) for all queries. The percentile parameter of our risk-weighting method is set to $p = 0.95$. We show percentiles (p-50, p-75, p-95, p-99, max) of the absolute execution time differences and the total relative slowdown compared to the best plan for both methods and all assumed and actual selectivities. In this experiment, the actual selectivities are picked around the estimated selectivity using a truncated normal distribution. We see that the minimum estimated cost plans are as good as or only slightly worse than the best plan in at least 75% of all simulated executions. However for some queries, 1% of the plans are significantly slower than the best plan choice (Q2: 165 s, Q10: 42 s, Q21: 901 s). Looking at all simulated executions of query Q2, plans chosen using the minimum estimated cost method are in total 5.68 times slower than the respective best plans

	Exec. time diff. MCP and BP (s)					slow	Exec. time diff. RCP and BP (s)					slow
	p-50	p-75	p-95	p-99	max	down	p-50	p-75	p-95	p-99	max	down
Q2	0	1	49	165	663	5.68x	0	0	0	14	18	1.23x
Q3	4	7	11	25	186	1.16x	4	6	16	26	32	1.16x
Q4	0	0	20	27	30	1.45x	0	0	16	18	21	1.25x
Q5	5	7	12	13	127	1.29x	6	8	11	13	21	1.35x
Q7	0	2	4	4	11	1.09x	0	2	15	25	29	1.20x
Q8	9	10	13	21	62	1.32x	0	8	11	13	23	1.13x
Q9	0	1	13	15	32	1.09x	0	1	13	14	16	1.07x
Q10	8	10	21	42	140	1.47x	9	11	22	28	33	1.54x
Q11	0	0	0	0	1	1.01x	0	0	0	0	0	1.01x
Q12	0	0	9	23	121	1.10x	0	0	0	8	18	1.01x
Q14	0	0	12	31	160	1.03x	0	0	0	5	9	1.00x
Q16	0	0	1	1	2	1.08x	0	1	3	3	4	1.20x
Q17	0	0	8	10	46	1.05x	0	0	7	10	11	1.04x
Q21	0	1	24	901	3593	2.10x	0	0	7	24	29	1.05x
ALL	0	4	13	37	3593	1.26x	0	2	11	20	33	1.12x

Table 5.2: Execution time difference of minimum estimated cost plan (MCP) and best plan (BP) and risk-weighted cost plan (RCP) and best plan. Actual selectivity is normal distributed around estimated selectivity ($\sigma = 10\%$) truncated between 0% and 100%, percentile parameter of risk-weighting method is $p = 0.95$.

(2.10x for query Q21 and 1.47x for query Q10). In contrast, the maximum absolute difference in execution time is much lower for plans chosen by the risk-weighted method (29 vs. 3593 seconds for Q21, 18 vs. 663 seconds for Q2) as well as the total relative slowdown (1.23x vs. 5.68x for Q2, 1.05x vs. 2.10x for Q21). However, there are also queries for which the risk-weighted method performs worse than the minimum estimated cost method, such as queries Q5, Q7, Q10, and Q16.

In order to evaluate how much risk-weighted plan choices improve or hurt the performance with respect to minimum estimated cost plan choices, we compare the execution times of plans chosen by both methods in more detail. Table 5.3 shows percentiles (min, p-1, p-5, p-95, p-99, max) of the relative and absolute differences of their execution times. We see that in several cases the execution time of risk-weighted plans is significantly higher than the execution time of the minimum cost plans. For example, in more than one percent of all simulated executions of query Q21, the execution time of a RCP is more than a thousand times higher the execution time of the MCP. However, when looking at the absolute difference, we find that the absolute regression is limited to 29 seconds while the maximum improvement is 3593 seconds. For query Q16 the maximum regression of risk-weighted plans is a factor of 225x but only 4 seconds. Over all plans, the maximum absolute regression is 32 second while for 7 queries the best improvement is more than 120 seconds. These observations show that risk-weighted plan choices prevent the execution of plans with extremely poor performance in worst-case scenarios at the cost of plan choices which are able to execute fast under favorable conditions.

	Relative Difference: $t(\text{RCP})/t(\text{MCP})$						Absolute Difference (s): $t(\text{RCP}) - t(\text{MCP})$					
	min	p-1	p-5	p-95	p-99	max	min	p-1	p-5	p-95	p-99	max
Q2	0.01x	0.02x	0.04x	2.21x	7.68x	35x	-663	-165	-49	0	0	1
Q3	0.23x	0.47x	0.85x	1.46x	2.73x	8.09x	-186	-24	-6	11	22	32
Q4	0.17x	0.18x	0.28x	1.79x	2.01x	2.09x	-30	-27	-17	3	4	4
Q5	0.24x	0.52x	0.89x	1.56x	1.60x	2.41x	-123	-6	-2	7	10	15
Q7	0.60x	0.71x	0.92x	1.62x	3.56x	8.29x	-9	-2	-1	12	24	29
Q8	0.28x	0.53x	0.67x	1.00x	1.01x	1.79x	-62	-21	-13	0	0	9
Q9	0.69x	0.74x	0.85x	1.02x	1.06x	1.37x	-20	-12	-5	0	2	13
Q10	0.24x	0.55x	0.76x	1.63x	3.30x	7.60x	-128	-27	-8	11	25	32
Q11	0.76x	0.91x	0.97x	1.01x	1.01x	1.04x	-1	0	0	0	0	0
Q12	0.18x	0.47x	0.72x	1.00x	1.24x	3.71x	-121	-23	-9	0	4	18
Q14	0.28x	0.49x	0.70x	1.00x	1.00x	1.77x	-160	-31	-12	0	0	9
Q16	0.82x	0.95x	0.98x	4.71x	47x	225x	-2	0	0	2	3	4
Q17	0.48x	0.73x	0.86x	1.01x	1.17x	1.35x	-46	-10	-5	0	5	10
Q21	0.01x	0.03x	0.54x	1.04x	1030x	1237x	-3593	-901	-23	0	24	29
ALL	0.01x	0.10x	0.69x	1.51x	3.54x	1237x	-3593	-35	-9	4	14	32

Table 5.3: Execution time difference of minimum estimated cost plan (MCP) and risk-weighted cost plan (RCP). Actual selectivity is normal distributed around estimated selectivity ($\sigma = 10\%$) truncated between 0% and 100%, percentile parameter of risk-weighting method is $p = 0.95$.

	Exec. time diff. MCP and BP (s)					slow	Exec. time diff. RCP and BP (s)					slow
	p-50	p-75	p-95	p-99	max	down	p-50	p-75	p-95	p-99	max	down
Q2	0	1	49	165	663	5.68x	0	0	3	15	24	1.34x
Q3	4	7	11	25	186	1.16x	4	8	14	20	32	1.17x
Q4	0	0	20	27	30	1.45x	0	0	17	25	28	1.35x
Q5	5	7	12	13	127	1.29x	6	8	10	13	20	1.35x
Q7	0	2	4	4	11	1.09x	0	2	4	4	11	1.09x
Q8	9	10	13	21	62	1.32x	0	8	12	13	17	1.15x
Q9	0	1	13	15	32	1.09x	0	1	13	14	16	1.07x
Q10	8	10	21	42	140	1.47x	8	10	12	20	27	1.45x
Q11	0	0	0	0	1	1.01x	0	0	0	0	0	1.01x
Q12	0	0	9	23	121	1.10x	0	0	7	10	18	1.05x
Q14	0	0	12	31	160	1.03x	0	0	0	5	9	1.00x
Q16	0	0	1	1	2	1.08x	0	1	1	1	2	1.14x
Q17	0	0	8	10	46	1.05x	0	0	6	10	11	1.04x
Q21	0	1	24	901	3593	2.10x	0	0	3	24	29	1.04x
ALL	0	4	13	37	3593	1.26x	0	3	10	16	32	1.11x

Table 5.4: Execution time difference of minimum estimated cost plan (MCP) and best plan (BP) and risk-weighted cost plan (RCP) and best plan. Actual selectivity is normal distributed around estimated selectivity ($\sigma = 10\%$) truncated between 0% and 100%, percentile parameter of risk-weighting method is $p = 0.70$.

	Relative Difference: t(RCP)/t(MCP)						Absolute Difference (s): t(RCP) − t(MCP)					
	min	p-1	p-5	p-95	p-99	max	min	p-1	p-5	p-95	p-99	max
Q2	0.01x	0.02x	0.04x	2.00x	6.30x	35x	-663	-165	-49	0	0	1
Q3	0.23x	0.47x	0.86x	1.47x	1.75x	4.04x	-186	-24	-6	12	15	28
Q4	0.17x	0.18x	0.98x	1.02x	1.95x	2.09x	-30	-26	0	0	4	4
Q5	0.24x	0.52x	0.89x	1.56x	1.60x	2.41x	-123	-6	-2	7	10	15
Q7	0.61x	0.71x	0.94x	1.08x	1.16x	1.42x	-3	-2	0	1	3	4
Q8	0.28x	0.53x	0.67x	1.00x	1.01x	2.42x	-62	-21	-13	0	0	16
Q9	0.68x	0.74x	0.86x	1.02x	1.06x	1.31x	-21	-12	-5	0	1	11
Q10	0.24x	0.51x	0.69x	1.49x	1.59x	2.26x	-128	-31	-11	9	11	18
Q11	0.76x	0.91x	0.97x	1.01x	1.01x	1.03x	-1	0	0	0	0	0
Q12	0.18x	0.51x	0.78x	1.00x	1.00x	2.21x	-121	-22	-7	0	0	10
Q14	0.28x	0.49x	0.70x	1.00x	1.00x	1.77x	-160	-31	-12	0	0	9
Q16	0.82x	0.95x	0.99x	2.96x	41x	63x	-2	0	0	1	1	1
Q17	0.48x	0.73x	0.86x	1.00x	1.16x	1.37x	-46	-10	-5	0	5	10
Q21	0.01x	0.03x	0.54x	1.00x	446x	1213x	-3590	-901	-23	0	24	29
ALL	0.01x	0.11x	0.70x	1.35x	2.18x	1213x	-3590	-35	-9	1	9	29

Table 5.5: Execution time difference of minimum estimated cost plan (MCP) and risk-weighted cost plan (RCP). Actual selectivity is normal distributed around estimated selectivity ($\sigma = 10\%$) truncated between 0% and 100%, percentile parameter of risk-weighting method is $p = 0.70$.

The performance-robustness trade-off of the risk-weighted plan choice method can be controlled by the percentile parameter. Table 5.4 and Table 5.5 show the differences in execution time if the percentile parameter of the risk-weighting method is set to $p = 0.70$ instead of $p = 0.95$. Comparing Table 5.4 with Table 5.2 we see that the effect of changing the percentile parameter differs among queries. While the worst-case execution times and relative slowdowns for some queries (Q2, Q4, Q12) increase, the plan choices for other queries (Q7, Q8, Q10, Q16) significantly improve. For example the maximum execution time difference of the RCP to the best plan is reduced from 29 to 11 seconds for query Q7. Table 5.5 shows that the maximum execution time regression (relative and absolute) of RCPs compared to MCPs for a percentile parameter of $p = 0.70$ is reduced for several queries (Q3, Q7, Q10, Q16) compared to a parameter setting of $p = 0.95$ (Table 5.3). At the same time, the gains in worst-case situations remain stable among all queries. These observations show that the optimal choice of the percentile parameter is not only application, but also query and workload specific.

The simulations so far were conducted for a scenario where the actual selectivity was picked from a truncated normal distribution that is centered on estimated selectivity with a standard deviation of 10%. In the following we show results for a simulation with a more pessimistic scenario where we pick the actual selectivity from a uniform distribution between 0% and 100%. Hence, the probability of the actual selectivity is independent from the estimated selectivity, which results in high over- and underestimations. We perform this simulation with a percentile parameter of $p = 0.70$ and show the results in Table 5.6 and Table 5.7. As expected, the execution time differences of minimum cost plans compared to the optimal plan choices significantly

	Exec. time diff. MCP and BP (s)					slow	Exec. time diff. RCP and BP (s)					slow
	p-50	p-75	p-95	p-99	max	down	p-50	p-75	p-95	p-99	max	down
Q2	0	12	605	1542	1851	34.35x	0	0	18	70	93	2.28x
Q3	4	9	26	222	600	1.33x	3	6	14	18	30	1.14x
Q4	0	0	25	31	44	1.51x	0	0	18	27	33	1.36x
Q5	6	9	17	69	253	1.38x	8	11	13	24	33	1.39x
Q7	1	3	9	18	50	1.16x	1	3	9	12	28	1.15x
Q8	10	12	17	112	195	1.45x	0	9	12	14	19	1.15x
Q9	0	3	13	32	59	1.11x	0	1	12	14	17	1.08x
Q10	9	16	75	241	412	1.96x	10	13	21	27	32	1.53x
Q11	0	0	1	2	3	1.04x	0	0	0	0	0	1.01x
Q12	0	0	47	185	375	1.38x	0	0	11	20	28	1.08x
Q14	0	0	31	144	425	1.07x	0	0	0	2	5	1.00x
Q16	0	1	3	6	10	1.15x	0	0	1	1	4	1.06x
Q17	0	0	15	89	151	1.12x	0	0	6	10	11	1.03x
Q21	0	2	204	7159	10359	8.54x	0	1	3	13	24	1.04x
ALL	0	6	28	350	10359	2.14x	0	3	13	22	93	1.12x

Table 5.6: Execution time difference of minimum estimated cost plan (MCP) and best plan (BP) and risk-weighted cost plan (RCP) and best plan. Actual selectivity is uniformly distributed between 0% and 100%, percentile parameter of risk-weighting method is $p = 0.70$.

	Relative Difference: $t(\text{RCP})/t(\text{MCP})$						Absolute Difference (s): $t(\text{RCP}) - t(\text{MCP})$					
	min	p-1	p-5	p-95	p-99	max	min	p-1	p-5	p-95	p-99	max
Q2	0.01x	0.01x	0.01x	1.05x	2.06x	23x	-1851	-1542	-605	0	0	2
Q3	0.09x	0.18x	0.50x	1.21x	1.45x	2.98x	-600	-220	-21	9	13	24
Q4	0.14x	0.17x	0.85x	1.03x	1.65x	2.08x	-44	-30	-1	0	3	4
Q5	0.14x	0.37x	0.81x	1.49x	1.57x	4.19x	-248	-64	-4	12	13	28
Q7	0.48x	0.61x	0.95x	1.12x	1.15x	1.18x	-41	-4	-1	1	4	5
Q8	0.11x	0.19x	0.59x	1.00x	1.00x	1.71x	-195	-111	-17	0	0	11
Q9	0.47x	0.66x	0.75x	1.04x	1.25x	1.46x	-49	-21	-11	1	8	15
Q10	0.09x	0.18x	0.40x	1.40x	1.49x	2.04x	-399	-223	-60	10	12	14
Q11	0.52x	0.62x	0.78x	1.01x	1.01x	1.02x	-3	-2	0	0	0	0
Q12	0.06x	0.12x	0.47x	1.00x	1.00x	1.28x	-375	-185	-39	0	0	5
Q14	0.11x	0.23x	0.61x	1.00x	1.00x	1.28x	-425	-144	-31	0	0	4
Q16	0.62x	0.68x	0.76x	1.28x	2.34x	59x	-10	-6	-3	0	1	1
Q17	0.22x	0.32x	0.63x	1.00x	1.01x	1.36x	-151	-89	-15	0	0	10
Q21	0.00x	0.01x	0.13x	1.00x	1.69x	1178x	-10346	-7158	-202	0	7	24
ALL	0.00x	0.02x	0.53x	1.06x	1.49x	1178x	-10346	-348	-19	0	11	28

Table 5.7: Execution time difference of minimum estimated cost plan (MCP) and risk-weighted cost plan (RCP). Actual selectivity is uniformly distributed between 0% and 100%, percentile parameter of risk-weighting method is $p = 0.70$.

	Exec. time diff. MCP and BP (s)					slow	Exec. time diff. BAP and BP (s)					slow
	p-50	p-75	p-95	p-99	max	down	p-50	p-75	p-95	p-99	max	down
Q2	0	1	49	165	663	5.68x	0	0	7	36	185	1.93x
Q3	4	7	11	25	186	1.16x	4	6	16	26	32	1.16x
Q4	0	0	20	27	30	1.45x	0	0	16	18	21	1.23x
Q5	5	7	12	13	127	1.29x	6	8	10	13	20	1.32x
Q7	0	2	4	4	11	1.09x	0	2	4	4	11	1.09x
Q8	9	10	13	21	62	1.32x	9	10	12	13	24	1.29x
Q9	0	1	13	15	32	1.09x	0	1	13	14	15	1.08x
Q10	8	10	21	42	140	1.47x	8	10	16	28	117	1.43x
Q11	0	0	0	0	1	1.01x	0	0	0	0	0	1.01x
Q12	0	0	9	23	121	1.10x	0	0	0	8	18	1.01x
Q14	0	0	12	31	160	1.03x	0	0	0	5	9	1.00x
Q16	0	0	1	1	2	1.08x	0	1	1	1	2	1.14x
Q17	0	0	8	10	46	1.05x	0	0	7	10	11	1.04x
Q21	0	1	24	901	3593	2.10x	0	0	2	10	29	1.03x
ALL	0	4	13	37	3593	1.26x	0	3	10	17	185	1.12x

Table 5.8: Execution time difference of minimum estimated cost plan (MCP) and best plan (BP) and Babcock plan (BAP) and best plan. Actual selectivity is normal distributed around estimated selectivity ($\sigma = 10\%$) truncated between 0% and 100%, percentile parameter of Babcock method [17] is $p = 0.70$.

increases compared to the previous simulations (Table 5.6, left-hand side). For 9 out for 14 queries, our simulation shows plan choices for which the difference in execution time to the optimal plan is more than 150 seconds. Similarly, the relative slowdown of minimum cost plans compared to the optimal plan increases (e. g., 34.35x vs. 5.68x for query Q2, 8.54x vs. 2.10x for query Q21). When comparing the execution time difference of risk-weighted plan choices (Table 5.6, right-hand side), we see that the maximum difference to the optimal plan increases only moderately compared to the simulation where actual selectivities were picked relative to the estimated selectivity (Table 5.4). The comparison of the relative and absolute execution time differences of minimum cost plans and risk score weighted plan choices (Table 5.7) shows that cases where RCPs perform worse than MCPs are similar to the less pessimistic simulations (Table 5.5), i. e., the risk-weighting plan choice method does not perform worse than the minimum cost method. At the same time, risk-weighted plan choices perform much better in worst-case situations compared to plan choices of the minimum cost method. The analysis of this simulation indicates that our method is rather insensitive to the distribution of actual execution conditions. However, an in-depth analysis is necessary to further validate this claim.

Our risk-weighting plan choice method is similar to a method proposed by Babcock et al. [17], which aims to improve the robustness of plan choices for queries with uncertain predicate selectivities. This method requires knowledge of the selectivity distribution of a predicate and uses a user-specified parameter to obtain a percentile value from the selectivity distribution similar to the percentile parameter of our risk-weighting method. The resulting selectivity value is used as a point estimate during plan enumeration and selection. Babcock et al. obtain the selectivity

	Relative Difference: $t(\text{BAP})/t(\text{MCP})$						Absolute Difference (s): $t(\text{BAP})-t(\text{MCP})$					
	min	p-1	p-5	p-95	p-99	max	min	p-1	p-5	p-95	p-99	max
Q2	0.01x	0.02x	0.05x	1.27x	6.36x	35x	-663	-164	-40	0	0	1
Q3	0.23x	0.47x	0.85x	1.46x	2.73x	8.09x	-186	-24	-6	11	22	32
Q4	0.17x	0.18x	0.28x	1.02x	1.95x	2.09x	-30	-27	-17	0	4	4
Q5	0.24x	0.52x	0.88x	1.54x	1.59x	2.41x	-123	-6	-2	7	9	16
Q7	0.60x	0.71x	0.92x	1.08x	1.15x	1.41x	-3	-2	0	1	3	4
Q8	0.28x	0.57x	0.68x	1.40x	1.45x	2.47x	-62	-18	-11	9	11	17
Q9	0.58x	0.76x	0.90x	1.02x	1.26x	1.46x	-27	-11	-4	0	9	15
Q10	0.17x	0.51x	0.70x	1.53x	1.62x	2.26x	-140	-29	-11	9	12	54
Q11	0.76x	0.90x	0.97x	1.01x	1.01x	1.03x	-1	0	0	0	0	0
Q12	0.17x	0.47x	0.72x	1.00x	1.24x	3.71x	-121	-23	-9	0	4	18
Q14	0.28x	0.49x	0.70x	1.00x	1.00x	1.77x	-160	-31	-12	0	0	9
Q16	0.82x	0.95x	0.99x	2.96x	41x	63x	-2	0	0	1	1	1
Q17	0.48x	0.73x	0.85x	1.01x	1.17x	1.37x	-46	-10	-5	0	5	10
Q21	0.00x	0.03x	0.52x	1.00x	1.09x	1237x	-3593	-901	-23	0	1	29
ALL	0.00x	0.16x	0.73x	1.32x	2.01x	1237x	-3593	-31	-8	1	10	54

Table 5.9: Execution time difference of minimum estimated cost plan (MCP) and Babcock plan (BAP). Actual selectivity is normal distributed around estimated selectivity ($\sigma = 10\%$) truncated between 0% and 100%, percentile parameter of Babcock method [17] is $p = 0.70$.

distribution by analyzing a sample of the queried base data set. In contrast, our method is not restricted to uncertain selectivities and includes the probability distribution of varying parameters through the risk score. We simulate the plan choices of Babcock's method and compare the results of this method with our method. We provide Babcock's method with precise information about the distribution of actual selectivities (normal distribution around the estimated selectivity with standard deviation $\sigma = 10\%$ truncated between 0% and 100%) and use a percentile parameter of $p = 0.70$[2]. Table 5.8 shows the execution time difference of Babcock plans (BAP) and the optimal plan. When comparing Table 5.8 with Table 5.4, we see that both plan choice methods behave similar for most queries. For queries Q4 and Q12, Babcock's method gives better results, while for queries Q2 and Q8 the plans chosen by our method perform better. Looking at the differences to the minimum cost plan choices in Tables 5.9 and 5.5, shows that both methods are sensitive to the same performance-robustness trade-off. For some queries (Q2, Q16, Q21), both methods choose plans that perform much worse than the minimum cost plans while preventing extremely risky plan choices.

We evaluated and compared different plan choice methods by simulating plan choices and executions with the help of the data we obtained from our performance study. The analysis of the simulation results showed that our risk score weighting plan choice method is able to prevent poor plan choices that cause exceptional high execution times. This comes at the cost of higher plan execution times in presence of favorable execution conditions. A comparison of our

[2]Note that the percentile parameters of our method and Babcock's method are not related and cannot be compared.

method with Babcock's approach suggests that the additional information of Babcock's method, i. e., exact knowledge of the selectivity distribution, does not significantly improve the quality of its plan choices.

5.3 Predicting Risk Scores for Execution Plans

In Section 5.1 we identified plan features that can cause significant performance variations in case of changing execution conditions. However, we also saw that the presence of a risky plan feature does not necessarily imply that the whole plan suffers from varying performance. Instead, the influence of a risky operator on the overall execution time and risk of a plan depends on several parameters, such as the size of its input data, the size of its workspace memory, and the size of the buffer pool. The problem of estimating these parameters falls back to the original problems of cardinality and memory estimation. But even if the performance variance of a single operator can be accurately estimated, it is still not trivial to infer the plan's overall variance of execution time. In principle, there are two aspects to consider when reasoning about the risk of a query plan. First, how much does the execution time of an individual operator vary for changing execution conditions? And second, how does the execution time of an operator relate to the overall execution time of the plan? An operator whose maximum costs are significantly larger than its minimum costs for changing execution conditions does not notably increase the overall risk of a plan if the operator's costs are negligible compared to the costs of the whole plan. Consequently, the overall risk of a plan is hard to quantify at optimization time. In this section we present an approach to assess the risk of a query execution plan using a machine-learned prediction model. We train a random forest regression model with the data we collected in the course of our experimental performance study (Section 5.1.3) to predict risk scores of execution plans.

In the following Section 5.3.1 we present our approach to predict the risk of relational execution plans using a random forest prediction model. We evaluate the predictive performance of our approach in Section 5.3.2 and analyze the importance of individual features in Section 5.3.3.

5.3.1 A Machine Learning Approach for Plan Risk Prediction

As summarized before, assessing the risk of a query execution plan is challenging. In this section, we present our approach to predict the risk of a query execution plan using a supervised machine learning technique. In the following we define the learning problem, describe how we derived the training data, present the feature vector design, and briefly introduce random forest regression models, which we used for our approach.

The Learning Problem We define the learning problem to predict the risk of a query execution plan as follows. Given a query execution plan, we predict the risk score of the plan as defined in Section 5.2.1, i. e., we predict the coefficient of variation of the plan's execution time for one specific distribution of execution conditions. We restrict the varying execution conditions to a single parameterized predicate, i. e., we assume that all other parameters of the execution

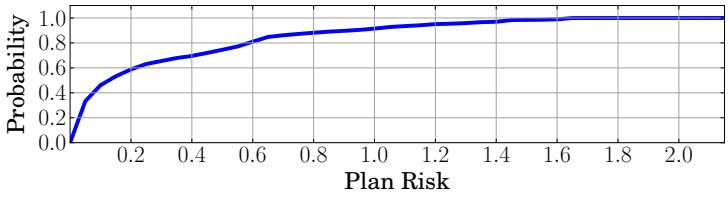


Figure 5.18: Cumulative value distribution of plan risk labels

condition (other predicates, memory, buffer pool state, and so on) remain constant. This problem statement includes risk predictions for queries with a single parameterized predicate, which are executed with varying parameter binding values under otherwise stable execution conditions.

The Training Data Set Our approach is based on a supervised learning technique and requires a data set to train a prediction model and test it. Each sample of a data set consists of a feature vector that represents the input of the prediction model and a prediction label, which defines the desired output of the model. For our application, a sample represents an execution plan and a partially fixed execution condition for which we want to predict the risk of the plan. Hence, a sample's feature vector should capture all relevant features of the plan and the fixed execution condition (such as amount of database memory) that determine the risk of the execution plan with respect to the variable part of the execution condition. The prediction label of a sample is the plan risk score for the given plan and partially fixed execution conditions with respect to a specific distribution of the variable predicate selectivity.

We derive our training and test samples from the data of the previously presented experimental performance study to analyze the performance variations of query execution plans (Section 5.1.3). When comparing the learning problem with the data collected for the study we see that the learning problem is restricted to a single variable predicate while the plan runs for the study were conducted with two varying predicates and varying database memory settings. The collected data provides 108 execution time measurements for each individual plan, one for each combination of two times six selectivity settings (one for each variable predicate) and three memory setting ($6 * 6 * 3 = 108$). From this data we compute 18 distinct training samples for each variable predicate of a plan. Each sample is specified by a plan, one of three memory settings (512, 1024, and 2048 MB), and one of six selectivity settings (0.1%, 1%, 5%, 10%, 50%, and 100% selectivity) for the non-variable predicate. Given 306 plans with two varied predicates, our training data set consists of 11016 samples in total.

The prediction label of the training samples is computed as the coefficient of variation of the six execution time measurements for the varied predicate selectivity. Figure 5.18 shows the cumulative value distribution of the prediction labels. The prediction label values range from 0.0 to 2.106 and follow a long-tail distribution. About 60% of the risk score labels are smaller than 0.2 and less than 10% of the samples have a plan risk of more than 1.0.

Note that we restrict the evaluation of our approach to a single probability distribution of uncertain execution condition, namely the distribution that was used for the experimental study. We would need to collect a new training data set in order to assess the risk of plans for a different distribution of execution conditions.

Feature Vector Design Careful design and selection of features is crucial to achieve high prediction performance with machine learning approaches. Usually, this task is laborious and requires extensive domain knowledge. While we are not aware of any other approach that tried to assess the performance variation of query execution plans using machine learning techniques, there is prior work which uses machine learning models to predict for example the resource consumption or execution time of query plans. These approaches proposed different alternatives to model the problem and design feature vectors for query execution plans: 1) plan-level modeling [88], 2) operator-level modeling [150], and 3) hybrid plan-operator-level modeling [8]. In plan-level modeling, one machine learning model is trained using features that describe a full plan. Operator-level modeling is a more fine-grained approach, which trains separate models for all types of operators using operator-specific features. To obtain the prediction for a plan, the plan is decomposed into its individual operators and for each operator a prediction is obtained using the respective model. The prediction of the whole plan is computed from all operator predictions. The hybrid approach, trains models on subplans (which might consist also of a single operator) of a full plan and combines their predictions.

Due to the complex structure of query plans, plan-level models tend to require a high dimensional feature vector and bear the risk to be under- or overfitted depending on the coverage of features in the training data. Since feature vectors need to be of constant size, the length of a plan's feature vector may not depend on its complexity, such as the number of operators or the shape of the operator tree. Instead, operator information needs to be aggregated (for example grouped by operator type). This can result in loss of valuable information. The plan-level approach is used by Ganapathi et al. [88]. In contrast, operator-level models are restricted to individual operators and can be trained with operator-specific features vectors. However, depending on the learning problem, it can be challenging to compute the final prediction for a plan from the predictions of its individual operators. For the prediction of resource consumption, operator-level predictions can be summed up [150], while execution time prediction would be more difficult because concurrently running operators due to pipelining and task parallelism need to be considered. In addition to the problem of computing the final prediction, operator-level modeling requires training labels on the level of individual operators.

Our approach to predict plan risk is based on a plan-level model. This is due to two reasons. First, the benchmark data includes only end-to-end execution times of plans, i. e., execution times on operator-level are not available. Second, it is not trivial to compute the risk of a plan from the risk of its individual operators as discussed before. Hence, we train a single model with a feature vector that contains the risk-relevant features of a whole plan. In the process of feature

vector design, we experimented with different features and feature compositions and trained and evaluated several prediction models. When designing feature vectors it is common that modifications do not have the same effect on all test samples but rather improve the prediction quality for some but worsen it for others. In this section, we report the result for the best feature vector that we found.

The findings of our experimental performance study (Section 5.1.3) influence the design of the feature vector. Our feature vector is composed of 127 features that can be grouped into three classes: 1) plan features, 2) features for operators with variable input aggregated by operator type, and 3) features for operators with constant input aggregated by operator type. Distinguishing between operators with variable and constant input sizes is a crucial aspect for the feature design. When predicting the risk score of the execution plan shown in Figure 5.13(b) with a parameterized predicate on the `Customer` relation (i. e., the predicate on `Supplier` is fixed) all operators below the lower `HashJoin` process a fixed amount of data, whereas the inputs of all following operators depend on the selectivity of the variable predicate on `Customer`. However, also operators with fixed input influence the risk of the plan because high constant plan costs reduce the relative execution time variance. We refer to the set of all operators of a plan p as O_p. The subset of operators with variable input cardinality is $O_p^v \subseteq O_p$ and the subset of operators with constant input cardinality is $O_p^c \subseteq O_p$. It holds $O_p^v \cap O_p^c = \emptyset$ and $O_p^v \cup O_p^c = O_p$. We refer with $O_{(p,t)}^v$ and $O_{(p,t)}^c$ to the set of all operators of operator type t in plan p with variable and constant input cardinality, respectively.

According to the definition of our learning problem, variation of plan execution time may only be caused by varying operator input cardinalities. Hence, predicting the risk score of a plan by using only structural plan properties, such as type and order of operators, is not a promising approach. Instead, the input size and costs of operators need to be taken into account to yield satisfying results. Our approach includes features which are computed from estimated operator input sizes and costs. For each training sample (i. e., a plan with one predicate selectivity being fixed and constant memory setting), we call the optimizer once for the lowest (0.1%) and once for the highest (100%) selectivity of the variable predicate to obtain minimum and maximum cardinality and cost estimates for all operators of the plan. This is similar to Picasso's method to collect cost estimates for plan space diagrams [179]. We refer to the low selectivity cost of an operator o as c_o^- and to the high selectivity cost as c_o^+. The low and high input cardinalities of the i^{th} input of an operator o are denoted as $\text{card}_{(o,i)}^-$ and $\text{card}_{(o,i)}^+$, respectively. The low and high result cardinalities of a plan p is referred to as card_p^- and card_p^+. In addition to cost and cardinality information, the optimizer provides the estimated optimal amount of workspace memory to process the plan, i. e., the amount of memory that is required to keep all processed data in memory. We refer with m_{opt}^- and m_{opt}^+ to the optimal amount of memory required to execute the plan with 0.1% and 100% selectivity, respectively.

An important aspect of feature vector design is generalization for unseen data, i. e., the ability to represent unseen data with similar feature vectors as the training data. For example varying

value domains of features caused by plans that query significantly larger data sets than plans in the training set can pose a challenge. We address this issue by computing most features as ratios instead of absolute values. The general applicability of the trained model is also supported by the prediction value itself as the coefficient of variation is a relative measure for variance instead of the absolute variance or standard deviation. The following ratios are used to build the feature vector.

The max-min cost ratio of a set of operators captures the relative difference of maximum and minimum cost for a set of operators and is defined as

$$r^{(\text{max-min})}(X) = \frac{\sum_{x \in X} c_x^+}{\sum_{x \in X} c_x^-}, X \subseteq O_p. \tag{5.3}$$

The spread-planMin cost ratio captures the absolute difference of maximum and minimum cost for a set of operators with respect to the minimal cost of the whole plan, i. e., the cost of the plan for the smallest input size (selectivity 0.1%). It is defined as

$$r^{(\text{spread-planMin})}(X) = \frac{\sum_{x \in X} (c_x^+ - c_x^-)}{\sum_{o \in O_p} c_o^-}, X \subseteq O_p. \tag{5.4}$$

The min-planMin cost ratio computes for a set of operators how much the sum of their low selectivity costs accounts for the low selectivity costs of the whole plan and is defined as

$$r^{(\text{min-planMin})}(X) = \frac{\sum_{x \in X} c_x^-}{\sum_{o \in O_p} c_o^-}, X \subseteq O_p. \tag{5.5}$$

Equivalently, the max-planMax cost ratio computes for a set of operators how much the sum of their high selectivity costs accounts for the high selectivity costs of the whole plan and is defined as

$$r^{(\text{max-planMax})}(X) = \frac{\sum_{x \in X} c_x^+}{\sum_{o \in O_p} c_o^+}, X \subseteq O_p. \tag{5.6}$$

Given the plan- and operator-level estimates obtained from the optimizer and the ratio definitions above, Table 5.10 lists all plan-level features (numbered p1 to p15). The plan-level features address four aspects of the plan, 1) the number of operators (p1 to p3), cost ratios for subsets of plan operators (p4 to p9), information about the result cardinality of the plan (p10 to p12), and about the memory budget of the database and the plan (p13 to p15). The result cardinality features (p10 to p12) give a hint for the time it takes the client to retrieve a computed result. This time is included in the execution time measure for the performance study but is not considered

	Feature Description	Definition
p1	Abs. number of plan ops	$\lvert O_p \rvert$
p2	Rel. number of var. input ops	$\lvert O_p^v \rvert / \lvert O_p \rvert$
p3	Rel. number of const. input ops	$\lvert O_p^c \rvert / \lvert O_p \rvert$
p4	Max-MaxPlan cost ratio of var. input ops	$r^{(\text{max-maxPlan})}(O_p^v)$
p5	Min-MinPlan cost ratio of var. input ops	$r^{(\text{min-minPlan})}(O_p^v)$
p6	Max-MaxPlan cost ratio of const. input ops	$r^{(\text{max-maxPlan})}(O_p^c)$
p7	Min-MinPlan cost ratio of const. input ops	$r^{(\text{min-minPlan})}(O_p^c)$
p8	Min-Max cost ratio of var. input ops	$r^{(\text{max-min})}(O_p^v)$
p9	Spread-MinPlan cost ratio of var. input ops	$r^{(\text{spread-minPlan})}(O_p^v))$
p10	Result card. for 0.1% selectivity	card_p^-
p11	Result card. for 100% selectivity	card_p^+
p12	Ratio of min. and max. result cardinality	$\text{card}_p^+ / \text{card}_p^-$
p13	DBMS memory in MB	m_{dbms}
p14	Ratio of min. and max. optimal memory budget	m_{opt}^+ / m_{opt}^-
p15	Ratio of the max. optimal memory budget and the db memory setting	m_{opt}^+ / m_{dbms}

Table 5.10: Plan-level features

in the optimizer's cost estimates. For plans with extremely varying output cardinalities, the time to fetch the result can have a significant impact on the execution time variance and hence on the plan risk. Features p13 to p15 use the optimizer's memory estimates and aim to capture the plan's demand for memory.

Table 5.11 lists the features for operators with variable input. We use these features for operators with variable input of the following types: `ClusteredIndexScan`, `ClusteredIndexSeek`, `SecondaryIndexScan`, `SecondaryIndexSeek`, `ScalarComputation`, `Sort`, `DistinctSort`, `StreamAggregation`, and `MergeJoin`. For the remaining join operators, we distinguish inputs of variable and constant size. Hence, we have `HashJoin-VarBuild`, `HashJoin-VarProbe`, `IndexNestedLoopJoin-VarOuter`, `IndexNestedLoopJoin-VarInner`, `NestedLoopJoin-VarOuter`, and `NestedLoopJoin-VarInner`.

Feature v5 is added for each input of an operator, i. e., we add it twice for a `MergeJoin` but not at all for a `ClusteredIndexScan` because it does not have an input. Features v7 to v9 are specifically designed to capture the cost of the inner side of nested-loop-based joins and are hence only added for `NestedLoopJoin-VarOuter` and `IndexNestedLoopJoin-VarOuter` operator types. The following equations define features v7 to v9. Thereby, the iOps(o) gives all operators of the inner side of the nested-loop join operator o, and oCard(o) gives the cardinality

	Feature Description	Definition
v1	Abs. number of ops of type t	$\lvert O^v_{p,t}\rvert$
v2	Rel. number of ops of type t	$\lvert O^v_{p,t}\rvert / \lvert O_p\rvert$
v3	Max-Min cost ratio of all ops of type t	$r^{(\text{max-min})}(O^v_{p,t})$
v4	Spread-MinPlan cost ratio of all ops of type t	$r^{(\text{spread-minPlan})}(O^v_{p,t})$
Feature each input i of operator type t		
v5/6	Ratio of the sum of the min. and max. input cardinalities of the i^{th} input for all ops of type t	$\frac{\sum_{o\in O^v_{p,t}} \text{card}^+_{(o,i)}}{\sum_{o\in O^v_{p,t}} \text{card}^-_{(o,i)}}$
Features for nested-loop-based joins with varying outer input		
v7	The ratio of the sum of single execution costs of the inner sides of all nested-loop join ops of type t and the minimum cost of the whole plan.	see Equation 5.7
v8	Max-min cost ratio of the inner sides of all nested-loop join ops of type t	see Equation 5.8
v9	Spread-MinPlan cost ratio of the inner sides of all nested-loop join ops of type t	see Equation 5.9

Table 5.11: Features for operators with variable input and type t

of the outer side of the nested-loop join operator o.

$$\frac{\sum_{o\in O^v_{p,t}} \left(\left(\sum_{x\in \text{iOps(o)}} c_x\right) / \text{oCard}(o)\right)}{\sum_{o\in O_p} c^-_o} \tag{5.7}$$

Equation 5.7 computes for each nested-loop join operator with variable outer side the cost of a single execution of its inner side (total cost divided by cardinality of outer side), sums these costs for all nested-loop join operators, and divides it by the minimum cost of the whole plan.

$$r^{(\text{max-min})}(X), X = \left\{\text{iOps}(o) | o \in O^v_{p,t}\right\} \tag{5.8}$$

$$r^{(\text{spread-minPlan})}(X), X = \left\{\text{iOps}(o) | o \in O^v_{p,t}\right\} \tag{5.9}$$

Equations 5.8 and 5.9 compute the max-min and spread-minPlan ratios for all operators which are on an inner side of a nested-loop join with variable outer side.

In total we have four operator types with four features (v1 to v4), four operator types with five features (v1 to v5), five operator types with six features (v1 to v6), and two operator types with nine features (v1 to v9). This results in 84 features for operators with variable input. If a plan does not have an operator of a specific type, all corresponding features are set to 0.

	Feature Description	Definition
c1	Abs. number of ops of type t with const. input	$\lvert O^v_{p,t}\rvert$
c2	Rel. number of ops of type t with const. input	$\lvert O^v_{p,t}\rvert / \lvert O_p\rvert$
c3	Ratio of the sum of the cost of all ops of type t with const. input and the sum of the low costs of all operators with var. input	$\frac{\sum_{o \in O^c_{p,t}} (c_o)}{\sum_{o \in O^v_p} (c^-_o)}$
c4	Ratio of the sum of the cost of all ops of type t with const. input and the sum of the high costs of all operators with var. input	$\frac{\sum_{o \in O^c_{p,t}} (c_o)}{\sum_{o \in O^v_p} (c^+_o)}$

Table 5.12: Features for operators with constant input and type t

Finally, Table 5.12 shows the features for operators with constant input size. We add these features for a selected set of operator types, namely `ClusteredIndexScan`, `ClusteredIndexSeek`, `SecondaryIndexScan`, `SecondaryIndexSeek`, `StreamAggregate`, `ScalarComputing`, and `HashJoin`. Given four features for each of these seven operator types, we have 28 features for operators with constant input size. Again, if a plan does not contain an operator of a specific type we set all corresponding features to 0.

Summing up the features of all three classes we have 15 plan level features, 84 variable input operator features, and 28 constant input operator features, which yields 127 features in total.

Random Forest Regression Models We use a random forest regression model to predict risk scores of query execution plans. Random forests are an ensemble technique, which means that it derives its predictions by combining multiple prediction models. In case of random regression forests these models are regression trees. A random forest is a collection of N regression trees [37] which are independently trained. Regression trees are binary trees whose inner nodes and leaves represent split conditions and prediction labels, respectively. A regression tree is trained as follows. At the beginning all samples of the training set are assigned to the root node. A node determines a split condition to group its samples into two sets such that their labels are separated in a "good" way. A split condition is usually considered to be "good" if it produces equally sized subsets where labels of both sets are as different as possible. The split condition is based on one (or more) elements of the samples' feature vectors. The splitting node creates two child nodes and assigns each subset to one of its children. Nodes are split until a termination criterion is reached. There are several conditions for stopping the splitting including a maximum tree depth, a minimum number of samples in a node, and pure nodes (all samples have the same label). A trained regression tree predicts a label for a given feature vector by repeatedly evaluating the split conditions of inner nodes, starting at the root node and following their path down to a leaf node. Finally, the prediction label is computed as the arithmetic mean of the samples' labels which were assigned to the identified leaf node during training.

Random forests train multiple regression trees and introduce randomization into the training such that the regression trees differ. A random regression forest computes a prediction as the mean of the predictions of all its regression trees.

There are several variations of random forests that differ in how they add randomization to the training process of the regression trees. In traditional random forests [37] each tree is built with a random subset (bootstrap sample) of the available training data. When determining the split condition, a random subset of K attributes is considered. For each of these K attributes, an optimal split value is computed and the best of the K splitting conditions is chosen. AdaBoost random forests [86] evaluate the prediction performance of already built trees and try to improve the predictions for samples that perform the worst so far.

For our learning problem we used random forests built from extremely randomized trees [91], which provided the best prediction performance in our experiments. These trees are trained as follows. In contrast to the traditional method, each tree is built with the full training set instead of a sample. Again K attributes are randomly chosen from the feature vector. For each attribute, a split value is chosen at random, which is used to create a randomized split condition. Each of the resulting K split conditions is evaluated and the best condition is chosen. We used the extremely random tree regressor implementation from the Python machine learning library scikit-learn [184].

In contrast to machine learning models that aim to approximate a function, such as linear regression or gradient descent methods, the value range of random forest predictions is limited to observed training samples, i. e., by the minimum and maximum label in the training data. For our application, this is not a fundamental limitation since the value domain of the coefficient of variation is quite narrow as shown in Figure 5.18. Moreover, our training data contains some very risky plans with execution time differences of more than five orders of magnitude. An important property of random forests is their ability to capture complex and non-linear dependencies by consecutively splitting the feature space. The ensemble property of random forests leads low variation of predictions. Finally, random forests are rather easy to use due to the small number of parameters and tolerance with respect to input features.

Model Training Parameters The performance of many machine learning techniques depends to a large extent on parameters that need to be tuned. The scikit-learn implementation of extremely random tree forests provides the following parameters.

The parameter N is the number of trees that are trained for the forest. The time to train the model as well as the time to compute a model prediction depends linear on the number of trees. The prediction quality of random forests improves asymptotically for an increasing number of trees, i. e., the prediction performance of a model will not decrease for a higher number of trees but does only slightly improve at some point [37]. K is the number of randomly chosen attributes, which are considered during split condition generation. For regression problems, setting K to

the size of the feature vector has been found to provide the best results in many cases [91]. Our experiments supported this finding. Note, that setting this parameter to the size of the feature vector effectively removes one degree of randomization from the model. Further, there are three parameters which act as conditions to limit the splitting of nodes. The maximum tree depth $D_{\max}$ limits the number of tree levels. The minimum number of samples for a splittable tree node is $S_{\min}$. A node with fewer samples will not be split and becomes a leaf node. And finally the minimum number of samples that a new node must contain in order to be created $L_{\min}$. If a node cannot be split in a way such that both of its children have at least this minimum number of samples, it is not split and becomes a leaf node itself. Parameters $D_{\max}$, $S_{\min}$, and $L_{\min}$ influence the number of samples in the leaf nodes and hence the number of sample labels that are aggregated to compute the leaf's prediction label. Thereby, more samples result in stronger smoothing of outliers, which can improve or worsen the prediction of samples.

Finding a good set of parameter values for $D_{\max}$, $S_{\min}$, and $L_{\min}$ requires the training and comparison of several models. For the each evaluation run, we did several parameter tuning runs to determine good parameter settings.

5.3.2 Evaluation of Prediction Performance

When evaluating machine-learned prediction models it is common practice to split the available data into non-overlapping training and test data sets. This method helps to prevent biased evaluation due to model overfitting. Evaluations that do not follow this rule are usually not helpful because they do not measure how well a model generalizes for unseen data. Hence, a part of the available data needs to be reserved for evaluation purposes and cannot be used for model training. Because the predictive performance of machine-learned models significantly depends on the number of training samples, reducing the size of the training data set can be challenging for applications where it is expensive or laborious to obtain training data, such as in our case.

A common approach to mitigate the cutback of training data is k-fold cross-validation. With cross-validation, the available data is divided into k subsets. Each of the k subsets is used once as test data set on a model that was trained using the remaining $k-1$ subsets. The performance of the model is computed as the average of all test runs. We use cross-validation techniques and the following metrics to evaluate the prediction performance of our approach.

The root mean squared error (RMSE) measures the average difference of predictions and true values and is defined as

$$\mathrm{RMSE}(y,\hat{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}, \tag{5.10}$$

where n is the number of test samples, and y_i and $\hat{y}_i$ are the true and predicted value of the i^{th} test sample, respectively.

The coefficient of determination or R^2 score is a measure for how well the data fits the prediction model. It is defined as

$$R^2(y,\hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \tag{5.11}$$

with $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. An R^2 score of 1 indicates that the model perfectly fits the data, i. e., the true values of all samples are exactly predicted. A score in the range between 1 and 0 indicates that the model's predictions are better than using the average of the true values as a prediction. An R^2 score of less than 0 indicates that the predictions of the model are worse than the average of the true values.

For measuring the prediction error of an individual sample we use the absolute error (AE) defined as

$$\mathrm{AE}(y_i,\hat{y}_i) = |y_i - \hat{y}_i|. \tag{5.12}$$

Baseline: Optimizer Risk Prediction We compute the plan risk for each training sample using optimizer estimated costs as a baseline for the evaluation of the predictive performance of our random forest prediction models. For each training sample we retrieve six plan cost estimates from the optimizer, one for each of the six selectivity bindings of the variable predicate. From these cost estimates we compute the samples' risk score prediction as the coefficient of variation. We treat this value as a risk score value predicted by the optimizer.

Figure 5.19 shows a binned scatter plot of the actual and optimizer predicted plan risk values[3]. The red line indicates perfect predictions. The plan risk scores of samples below and above this line are under- and overestimated, respectively. The binned scatter plot of Figure 5.19 shows that the plan risk scores of samples with low plan risk are often accurately predicted by the optimizer. However, except for samples with low actual plan risk, the predicted risk scores are mostly imprecise and often overestimated. Figure 5.19 indicates that computing plan risk predictions from optimizer cost estimates does not yield satisfying results.

[3]Figure 5.19 differs from Figure 5.7 because it shows risk scores for plans with a single variable predicate.

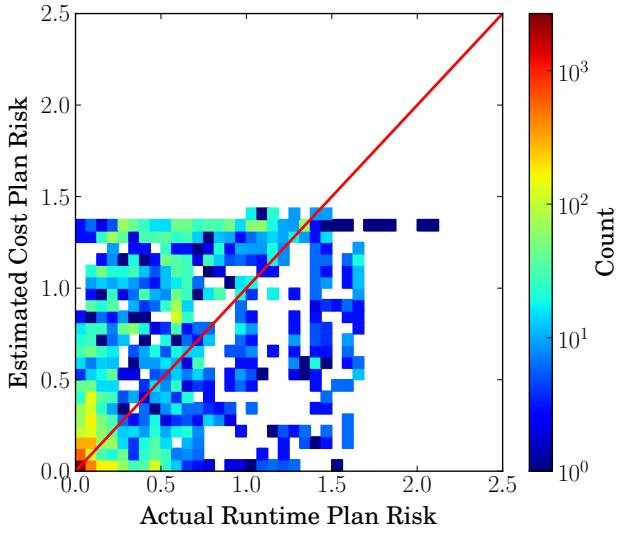


Figure 5.19: Actual execution time plan risk scores vs. plan risk scores predicted by optimizer cost estimates.

Random k-fold Cross-Validation Next, we evaluate the performance of a random forest prediction model with 10-fold cross-validation. We randomly split our training data set into ten subsets of about equal size. We train ten models with nine subsets and test with the remaining tenth set. Each model consists of 5000 trees with a maximum tree depth of 36 and the minimum sample split and minimum sample leaf parameters set to 4 and 2, respectively.

Figure 5.20 shows a binned scatter plot of actual and model predicted plan risk values. With an average RMSE of 0.0348 and an average R^2 score of 0.9921, the obtained results are too good are require a closer look. The extremely good prediction performance is caused by model overfitting. Each plan of our experimental study is represented by 36 samples in our data set (18 samples for each of both variable predicates). The ten randomly assembled cross-validation folds contain very similar samples which causes overfitted prediction models. Therefore, this performance evaluation is not conclusive.

Query-wise Cross-Validation To overcome the problem of model overfitting and to simulate the prediction of plan risk values for plans of unseen queries, we split the data set into 14 subsets, one set for each individual query. We run a cross-validation with 5000 trees (maximum tree depth 36, minimum sample split 20, and minimum sample leaf 10) for these query-grouped folds. Note that the model parameters for the minimum number of samples for split and leaf nodes are higher than for the random cross-validation run to achieve stronger smoothing of the predictions by averaging over more samples.

Figure 5.22 shows a binned scatter plot with the predictions of all cross-validation runs. Com-

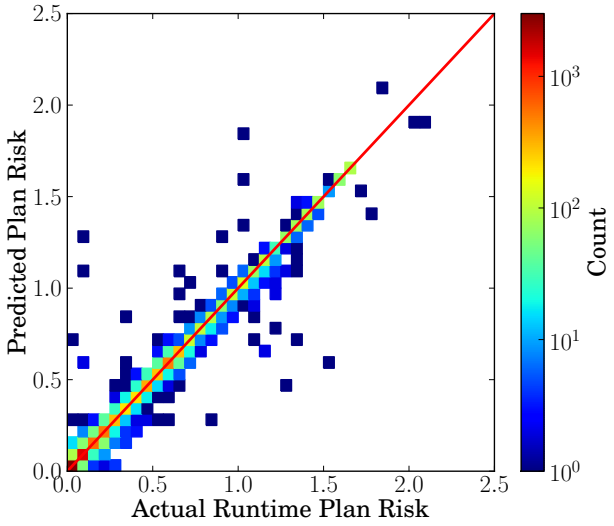


Figure 5.20: Actual and predicted plan risk scores for random 10-fold cross-validation.

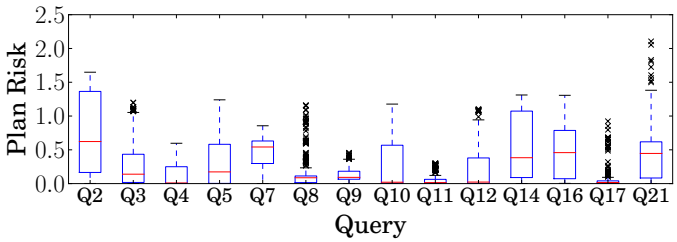


Figure 5.21: Distribution of actual plan risk scores by query.

pared to Figure 5.19 we see that the predictions for samples with an actual plan risk score between 0 and 0.8 are much closer distributed around the red line indicating perfect predictions. For samples with actual plan risk scores higher than 0.8 the model predictions are less precise and often significantly underestimated.

Table 5.13 shows the root mean squared errors and R^2 scores for all cross-validation runs and their mean values (weighted by the number of test samples per query). We also add the RMSE and R^2 scores of the optimizer predictions for comparison. Figure 5.23 shows a binned scatter plot for each query with actual and predicted plan risk values. Table 5.13 and Figure 5.23 show that the prediction performance of the regression model heavily varies among queries. While the risk scores for plans of some queries are quite accurately predicted, other queries suffer from significant prediction errors. Figure 5.21 shows the value distribution of plan risk scores grouped by query using box-and-whisker plots (blue crosses indicate outliers). When comparing Table 5.13 and Figure 5.21 we see that the prediction model has the lowest prediction errors for queries with plans that have low risk values, i. e., queries Q4, Q8, Q9, Q11, and Q17. In contrast

Query	#Samples	RMSE		R^2	
		RF	OPT	RF	OPT
Q2	1368	0.5656	0.4971	0.0525	0.2680
Q3	648	0.0991	0.4347	0.8905	-1.0189
Q4	504	0.0759	0.3644	0.7361	-5.0783
Q5	612	0.1391	0.4146	0.8197	-0.5124
Q7	864	0.2398	0.4151	-0.0649	-2.1910
Q8	1332	0.0912	0.3102	0.7235	-2.1970
Q9	1692	0.0773	0.3200	0.1748	-13.1236
Q10	756	0.1885	0.2249	0.6945	0.5649
Q11	576	0.0907	0.4516	-0.2302	-29.4607
Q12	288	0.1444	0.4104	0.7827	-0.7533
Q14	144	0.4222	0.5906	0.1832	-0.5981
Q16	1008	0.2394	0.3194	0.6232	0.3297
Q17	396	0.1292	0.3342	0.2844	-3.7879
Q21	828	0.4311	0.3766	0.0452	0.2200
Mean	-	0.2149	0.3733	0.3765	-4.3741

Table 5.13: Performance of query-wise cross-validation prediction (RF) compared to performance of optimizer estimated cost prediction (OPT).

the performance of the prediction model is much worse for queries with several high risk plans (Q2, Q7, Q14, Q16, and Q21). This finding is also supported by Figure 5.22, which shows good prediction performance for samples with actual plan risk scores up to 0.8 and significantly higher errors for samples with risk scores of more than 0.8.

Figure 5.18 helps to explain the influence of a query's plan risk score distribution on the regression model's prediction performance. It shows that the actual plan risk values follow a long-tailed distribution with few plans having high risk scores. Only 10% of the samples have a plan risk score of 0.8 or more, only 5% of 1.2 or more. Therefore, the model might suffers from lack of training samples for high plan risk scores. This lack of training data is exacerbated by the fact that only few queries have samples with high plan risk values out of which one query can be held back as test set due to the cross-validation technique.

Looking at Table 5.13, we find that the R^2 scores correlate with the RMSE values in general, i. e., a low RMSE indicates a high R^2 score. There are a four queries for which this observation does not hold (Q7, Q9, Q11, and Q17). Three of these queries have samples with low plan risk scores (Figure 5.21).

Note, we do not find conclusive evidence that the model's prediction performance depends on the complexity of a query (and plan). The four queries with an RMSE of less than 0.1 access 3, 6, 6, and 8 relations while the four queries with an RMSE of more than 0.2 access 2, 6, 6, and 9 relations.

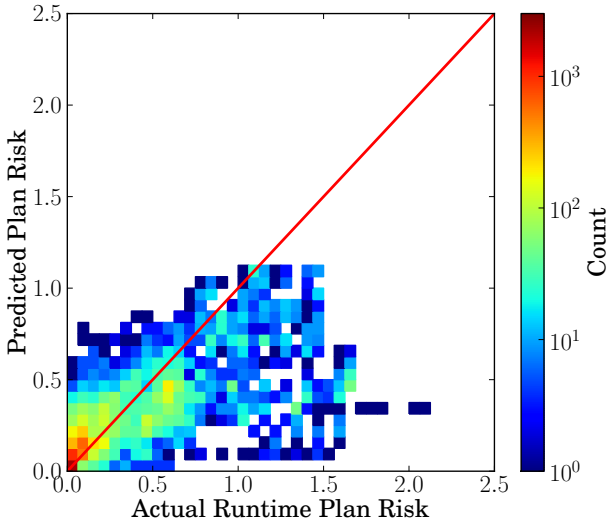


Figure 5.22: Actual and predicted plan risk scores for query-wise cross-validation.

Comparing the prediction performance of the regression model and the predictions computed from the optimizer's cost estimates, we see that the prediction errors of the optimizer predictions are more stable (RMSE between 0.22 and 0.59) than the errors of the model. Except for query Q2 and Q21, the random forest regression model outperforms the optimizer predictions in RMSE and R^2 score. In comparison to the optimizer's predictions, the regression model performs well for queries whose plans have mostly low risk scores.

Next, we analyze in detail why the regression model performs poorly for plans of certain queries. There are several reasons why a regression model may not be able to accurately predict the labels of a sample. As already mentioned, lack of training data can be a reason for the poor prediction performance. Another possibility is that the feature vector does not model characteristics that influence the label to predict. Further, the prediction performance of a model may suffer if some aspects that influence the label and which are actually represented in the feature vector are only present in a subset of samples that all belong to the same fold. For our use case this happens if a risky plan characteristic, such as a specific operator, is only present in plans of the same query. All these reasons can affect machine learning approaches in general. In addition, our approach has to deal with an additional challenge. Since most of our features are computed from optimizer estimates, the accuracy of optimizer estimates can be important for the performance of our prediction models.

In order to analyze why our prediction models perform poorly for plans of certain queries, we take a closer look at four queries with high RMSE prediction errors, namely Q2, Q10, Q14, and Q21. We start analyzing how the models' prediction errors correlate with the error of the optimizer's predictions. Figure 5.24 shows a binned scatter plot of the errors (AE) of the random forest models and the optimizer predictions.

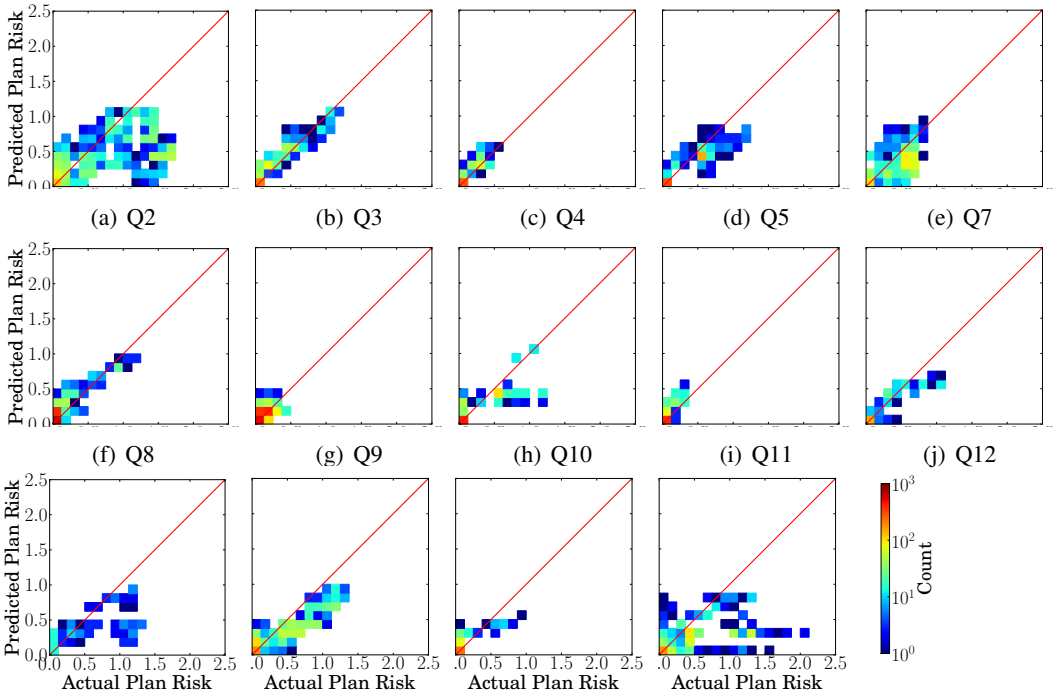


Figure 5.23: Actual and predicted plan risk scores for query-wise cross-validation by query.

For the discussion, we divide the plot into six areas as indicated by the dashed red lines. Area 1 in the bottom left corner includes samples for which both the regression model and the optimizer produce low estimation errors (AE ≤ 0.25). Areas 2 and 3 contain samples for which the regression model has low estimation errors and the optimizer has high errors (AE > 0.25) and vice versa. Area 4 includes samples for which the optimizer and the regression model produce similar high prediction errors. Areas 5 and 6 contain samples for which both, regression model and optimizer produce high errors, but where the error (AE) of the optimizer exceeds the error of the regression model by at least 0.25 and vice versa.

We see that most of the samples fall in Area 1 (56.8%) and Area 2 (26.2%). Area 3 contains about 7.3% of the samples. Figure 5.24 shows for 6.8% of the samples a correlation between the optimizer and the regression model prediction errors (Area 4). The high optimizer prediction errors indicate inaccurate cardinality or cost estimates. Since several features are derived from optimizer estimates, estimation errors might have been propagated into the model. Areas 5 and 6 contain only 1.0% and 1.9% of the samples.

Figure 5.25 shows the model and optimizer errors (AE) for all samples grouped by query. Similar to Figure 5.23 we see significant differences between individual queries. For several queries,

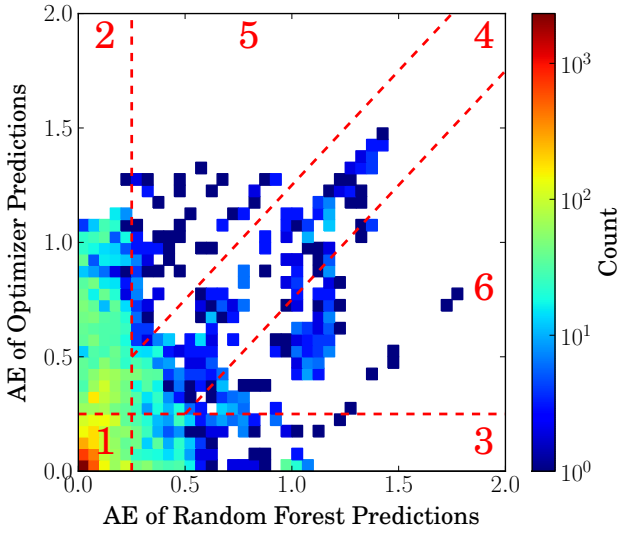


Figure 5.24: Absolute error (AE) of model predictions (query cross-validation) vs. AE of optimizer predictions.

including Q3, Q4, Q9, and Q11, the plots show that the random forest model is able to "correct" imprecise optimizer estimates. For the queries with the highest RMSE (Q2, Q14, and Q21) several samples are located in Area 4. This indicates that inaccurate optimizer estimates are propagated through feature vectors into the regression models and theirs predictions. The same queries have also several samples in Areas 3 and 6 indicating that the model produces worse predictions than the optimizer. Although, the regression model is often able to improve over optimizer predictions, we cannot expect that it provides accurate predictions even if the optimizer estimates are far off. However, the goal of the model should be to provide predictions that are not significantly worse than the optimizer's predictions. In the following, we analyze why the regression model produces bad predictions for the queries Q2, Q14, Q21, and Q10.

Query Q2 joins nine relations and includes a subquery, which is executed in some plans as an `IndexNestedLoopJoin` with multiple operators on the inner side and an outer side that depends on a variable predicate. The execution time of these plans significantly depends on the cardinality of the outer input of the `IndexNestedLoopJoin`. All samples of Q2 plans that feature the complex `IndexNestedLoopJoin` fall into Area 1 and 4 of Figure 5.25(a). This indicates that the optimizer is not able to correctly estimate the cost of this join for a range of selectivities. The plans of all samples in Area 3 and 6 have another pattern in common and include two `SecondaryIndexSeek` operators where the selectivity depends on the variable predicate. In our data set, only query Q2 features plans with multiple `SecondaryIndexSeek` operators. The model cannot learn this pattern when all Q2 samples are included in the test set as required by query-wise cross-validation.

Query Q14 is a simple query, which filters and joins two relations, `lineitem` and `part`. It differs

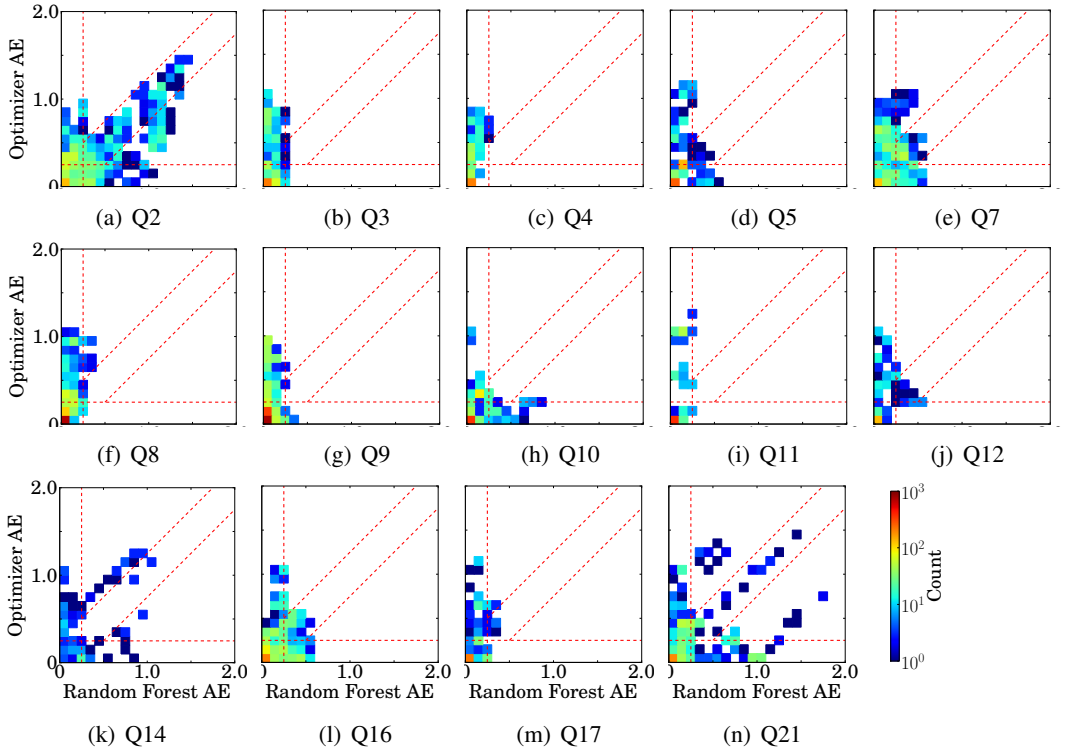


Figure 5.25: Absolute error (AE) of model predictions (query cross-validation) vs. AE of optimizer predictions by query.

from most other queries in our data set (except Q2) because it does not return an aggregated result. Instead its output cardinality ranges from 0 to 6 million tuples (the cardinality of the `lineitem` relation in a scale factor 1 TPC-H data set). For queries with rather low computational overhead (such as Q14), the time to retrieve the result by the client can account for a large fraction of the overall execution time. The optimizer does only give an estimate for the efforts to compute the result of a plan but does not take the time to fetch the result into account. Because Q14 is the only query with such a large output cardinality range, this pattern cannot be learned by the regression model (Query Q2's output cardinality depends on the much smaller `part` relation and Q2 is much more expensive to compute). Consequently, samples with extremely varying output cardinality, i. e., samples where the fixed predicate has a high selectivity, fall into Areas 4 and 5 of Figure 5.25(k). Figure 5.25(k) does also show that for some samples the optimizer's predictions are much better than the regression model's predictions (samples in Area 3). For these samples the predicate of the `lineitem` relation is varied. The corresponding execution plans read the `lineitem` relation using a `SecondaryIndexScan` and feed the result into the build side of a `HybridHashJoin`. While the optimizer correctly estimates the varying execution costs of these samples, it seems that the regression model does not capture the relevance of

the SecondaryIndexScan and HybridHashJoin for the risk score of the plan. This might be caused by lack of training data because the data set does not include other plans with just a single HybridHashJoin operator, which can be considered as rather robust compared to nested-loop based joins. Similar to query Q14, queries Q4 and Q12 also join two relations. However, these joins are performed on indexed attributes such that IndexNestedLoopJoin or MergeJoin operators are preferred.

Query Q21 joins four relations and includes an EXIST and a NOT EXIST subquery both referring to the large lineitem relation. Figure 5.25(n) shows that for some samples the regression model outperforms the optimizer (Areas 2 and 5). However, there are also samples for which the error of the regression model correlates with the optimizer's error (Area 4) or is much worse (Areas 3 and 6). Looking at samples for which the regression model produces predictions errors of more than 0.7 (RMSE), we find that all corresponding plans contain a NestedLoopJoin with a ClusteredIndexScan of the lineitem relation on the inner side, i. e., the full relation is scanned for each tuple (or set of tuples) of the outer input. The outer input of the NestedLoopJoin is the result of a join of the supplier and nation relations. The result of the NestedLoopJoin is fed into a cascade of three subsequent IndexNestedLoopJoins. In Q21, the variable local predicates are applied on the supplier and the lineitem relations. For samples where the supplier predicate is fixed (i. e., the outer input of the NestedLoopJoin is constant and the size of the inner input varies), the prediction errors of the regression model and the optimizer are similar (samples in Area 4). If the local predicate of the lineitem relation is fixed and the selectivity of the supplier predicate is varied, the optimizer predictions are much better than the predictions of the regression model (Areas 3 and 6). NestedLoopJoins with expensive inner sides occur only in 9 of the 306 benchmarked plan out of which 8 plans are for query Q21. Hence, the regression model can learn this property of risky plans only from one other plan when testing the model with samples from query Q21. In addition to inaccurate optimizer estimates, this lack of training samples contributes to the low performance of the regression model for some of the plans of query Q21.

Figure 5.25(h) shows that the regression model performance is worse than the optimizer predictions for some samples of query Q10. These samples correspond to three distinct plans, which are executed with the lowest database memory setting (512 MB). All plans exhibit the same property as plan Q10-2 presented in Section 5.1.3, i. e., they have a Sort operator in front of an IndexNestedLoopJoin that references a clustered index on lineitem on its inner side. This Sort operator shuffles the order of the join attribute causing massive random I/O due to index seek operations. Because of the low memory setting, only a small fraction of I/O operations can be served by the buffer pool. Our analysis indicates that the regression model does not capture the effect of (un-)sorted outer sides on IndexNestedLoopJoins. In fact, our feature vector does not distinguish between IndexNestedLoopJoin operators with sorted and unsorted outer sides. We experimented with adding corresponding features, however this did not considerably improve the predictions for samples of query Q10 and reduced the overall prediction performance of the model.

The detailed analysis of prediction errors for queries Q2, Q10, Q14, and Q21 indicates that high prediction errors of the regression model are mostly caused by inaccurate optimizer estimates or lack of appropriate training data for certain properties of risky plans. Inaccurate optimizer estimates are one of the main motivations to determine the risk score of a plan. Hence, it would be desirable to have a method to assess the risk of an execution plan that is not sensitive with respect to inaccurate optimizer estimates. In fact, the analysis of the prediction performance shows that the prediction model is often able to "correct" imprecise optimizer plan risk predictions. However, for 6.9% of the samples the errors of the optimizer and model are similar which indicates that inaccurate optimizer estimates are propagated into the model. In cases where the optimizer predictions significantly outperform the regression model (about 9.2% of our training samples), the detailed analysis of queries Q2, Q10, Q14, and Q21 shows that the model is not aware of specific risky plan features. Due to query-wise cross-validation, such risky plan features are only present in the test fold and cannot be learned during the training phase. Additional training data from more diverse queries that cover these risky plan features could improve this situation. Although our analysis indicates that lack of appropriate training data is the primary reason for poor prediction performance, we cannot preclude that inappropriate feature vector design plays a role as well.

Plan-wise Cross-Validation In order to ascertain that feature vector design is not a major reason for the insufficient prediction performance of our models, we train prediction models with additional samples by conducting another set of cross-validation runs. For these runs, we assign all samples with the same variable predicate of a plan to a separate fold resulting in 612 folds, each consisting of 18 samples (six selectivity settings times three memory settings). Since the risk score of a plan can differ significantly for different variable predicates, having the same plan with different variable predicates in separate folds should not cause model overfitting. A more noteworthy problem might be plans which are very similar, such as the plans discussed in Section 5.1.3. However, these examples demonstrate that even small changes of a plan, such as a reordered `Sort` operator, can cause large differences in the risk score. Due to the increased number of folds (612 vs. 14), we reduced the number of trees to 500. All other parameter are set to the same values as in the query-wise cross-validation experiment, i. e., a maximum tree depth of 36, at least 20 samples to split a node, and at least 10 samples for each leaf node.

Figure 5.26 shows a binned scatter plot of the predicted and actual plan risks for all samples. As expected, the prediction performance of regression models trained with plan-wise cross-validation is much better than the performance of the models trained with query-wise cross-validation (compare to Figure 5.22). Figure 5.27 displays binned scatter plots for the predicted and actual plan risks of queries with previously low prediction performance (Q2, Q10, Q14, and Q21). Comparing these plots with Figures 5.23(a), 5.23(h), 5.23(k), and 5.23(n), we see significant improvements for all queries. However, there are still a few outliers for which the risk is not accurately predicted. We account these inaccurate predictions to imprecise optimizer

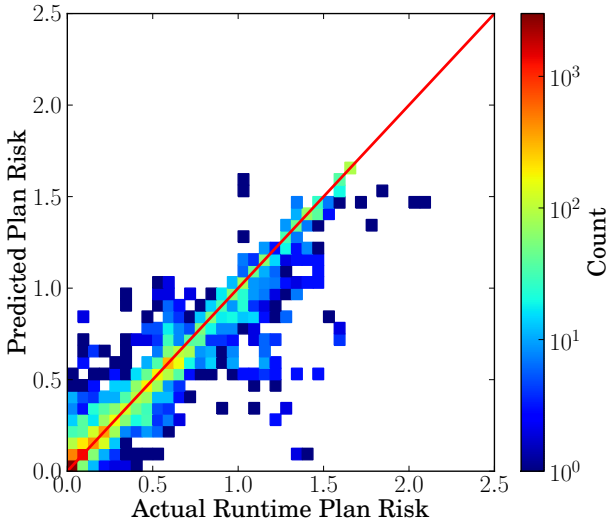
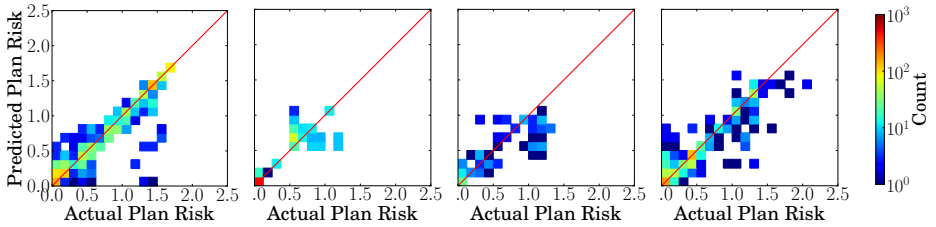


Figure 5.26: Actual and predicted plan risk for plan-wise cross-validation.



(a) Q2 (b) Q10 (c) Q14 (d) Q21

Figure 5.27: Actual and predicted plan risk for plan-wise cross-validation predictions by query.

estimates. This experiment supports the conclusion of our previous analysis that additional training data improves the prediction performance for plans with underrepresented risky plan features and indicates that the feature vector is able to represent also these infrequent risky plan features.

5.3.3 Analysis of Feature Importance

Due to their hierarchical structure and easy-to-understand split conditions, regression tree models have the nice property that they can be analyzed and understood by humans. This property differentiates decision trees from machine learning techniques that produce black box models as for instance support vector machines (SVMs). Given a regression tree model it is possible to assess for an individual feature how much it contributes to label prediction, i. e., how important it is. The importance of a feature can be computed by estimating the fraction of samples which are

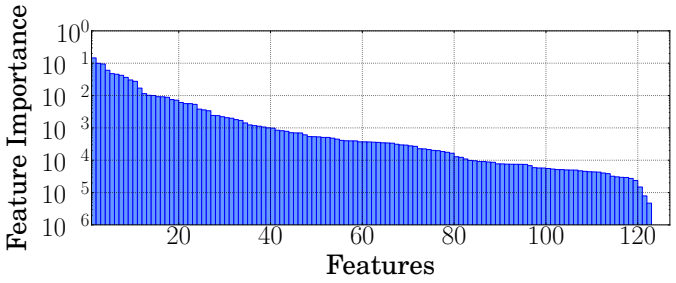


Figure 5.28: Distribution of feature importance for query-wise cross-validation runs.

partitioned by split conditions that evaluate the feature. A feature that is evaluated in multiple split conditions of nodes close to the root of a regression tree influences the predictions of more samples than a feature which is evaluated in a single node close to a leaf. Please note that the importance of a feature depends as well on its frequency in the training data set, i. e., a feature for an operator type that is rare but highly discriminating might have a lower importance score than a less discriminating but more frequent feature. Therefore, a feature importance analysis is specific for a trained model and not for the prediction problem in general.

The importance of a feature in a random forest model is computed by averaging its importance in all regression trees of which the model consists. The average computation also reduces the variance of the importance estimation induced by the random nature of the method. The scikit-learn [184] implementation of extremely random trees computes the relative importance of each feature. In the following we analyze the importance of features in a random forest model trained with query-wise cross-validation.

Figure 5.28 shows the importance of all 127 features ordered by importance on a logarithmic scale. The figure shows that the importance of features follows a long tail distribution. The top 20 features account for approximately 92.5% of the feature importance. The most important feature has a relative importance of 20.52%. Four features did not contribute to the predictions at all (0.0%). Next we discuss the most influential features.

Table 5.14 lists the 20 most important features. The most important feature is plan-level feature p8, i. e., the max-min cost ratio of all operators with variable input (see Equation 5.3). This feature determines how much the cost of all operators with changing input cardinalities varies for changing predicate selectivities. The third, fourth, and fifth most important features are also plan-level features. The third and fourth features are the max-maxPlan cost ratio (see Equation 5.6) of all operators with variable and constant input, respectively. The fifth feature is the spread-minPlan cost ratio of all operators with variable input. These three plan-level features model in different ways how much operators with variable and constant input sizes influence the overall cost of a plan. Further important plan-level features address the output cardinality of plans and are on rank 11 (p12, the ratio of the plan's output cardinality for minimum and maximum input size) and rank 18 (p11, the plan's output cardinality for maximum input size).

rank	Operator type / Plan	Feature ID	Importance
1	Plan	p8	20.52%
2	`SecondaryIndexSeek` op with var input	v1	14.56%
3	Plan	p4	9.93%
4	Plan	p6	9.46%
5	Plan	p9	6.05%
6	`ComputeScalar` op with const. input	c1	4.74%
7	`IndexNestedLoopJoin` op with var. outer input	v8	4.55%
8	`IndexNestedLoopJoin` op with var. outer input	v5	4.22%
9	`SecondaryIndexSeek` op with var. input	v3	3.64%
10	`ComputeScalar` op with const. input	c2	3.03%
11	Plan	p12	2.76%
12	`IndexNestedLoopJoin` op with var. outer side	v3	1.70%
13	`SecondaryIndexSeek` op with var. input	v4	1.15%
14	`Sort` op with var. input	v5	1.02%
15	`IndexNestedLoopJoin` op with var. outer side	v4	1.00%
16	`IndexNestedLoopJoin` op with var. outer side	v9	0.92%
17	`NestedLoopJoin` op with var. outer side	v7	0.92%
18	Plan	p11	0.88%
19	`ClusteredIndexSeek` op with var input	v4	0.75%
20	`NestedLoopJoin` op with var. outer side	v9	0.72%

Table 5.14: The 20 most important features for query-wise cross-validation. See Tables 5.10, 5.11, and 5.12 for feature descriptions.

Looking at operator-level features, the second most important feature is the absolute number of `SecondaryIndexSeek` operators with variable output in the plan (v1). The same operator is also represented by features v3 and v4 with ranks 9 and 13, respectively. Other operator types that appear in the top 20 features are `ComputeScalar`, `IndexNestedLoopJoin`, `NestedLoopJoin`, `Sort`, and `ClusteredIndexSeek`. Given the risky plan features we identified based on our experimental performance study (see Section 5.1.4), it is not surprising to find features of exactly these operator among the most important ones, except for the features of `ComputeScalar` operators with constant input size.

Table 5.15 lists for each operator type the number of features in the feature vector, the best ranking of a feature for this operator type, the average ranking, the maximum importance of a feature for this operator type, and the sum of the importance of all features for this operator type. It shows that plan-level features are the most important features and account for more than 50% of the relative importance. The ranking of the following operators is not unexpected. Features for operators with variable input are more important than features for operators with constant input. Among the operators with the most important features are index seek operators and nested-loop-based joins with variable outer input. As discussed before, both can cause significantly higher execution time for increasing input sizes or predicate selectivities. Features for `Sort` operators are relevant as well. The much lower importance of `DistinctSort` operator features compared to the regular `Sort` can be explained by their less frequent occurrence in our data set. Features for `ClusteredIndexScan` operators are also relevant for the prediction of plan risk as they tend to cause a high number of I/O operations, which can account for a large fraction of plan cost. Features for `SecondaryIndexScan` operators with variable output are not relevant for the model because none of the plans in our data set contains this operator. We also see that neither `HybridHashJoin` operator features, both for variable build and probe inputs, nor nested-loop based joins with variable inner sides are important for the plan risk prediction model.

There are also a few surprising observations. First of all, the high importance of `ComputeScalar` operator features, both for constant and variable inputs, is remarkable. This operator type does not cause any I/O operations and its cost should depend linearly on the input size. Hence, it is unlikely that this operators actually influences the risk of a plan. One explanation is that this feature is used by the model to distinguish plans of different queries. For example the plans of query Q2 are the only ones in our data set without any `ComputeScalar` operator. Another unexpected observation is that the features of `HybridHashJoin` operators with variable probe side are more relevant than features of `HybridHashJoin` operators with variable build side. This might be explained by the fact, that parts of the probe side are spilled to disk if the build side exceeds the memory budget. Since the probe side is usually larger than the build side, its spilling causes more I/O operations. Lack of memory for hash tables might frequently happen for low database memory settings. Comparing the frequency of these features, we find that 5760 `HybridHashJoin` operators with variable build side and 10008 ones with variable probe side are included our training data set. Hence, the difference in occurrence frequency might have

	Operator	Var/Const	#Features	Best Rank	Avg. Rank	Max Imp.	Σ Imp.
1	Plan	-	15	1	25.27	20.519 %	50.865%
2	SecondaryIndexSeek	var	4	2	11.75	14.564%	19.916%
3	IndexNestedLoopJoin (var. outer side)	var	9	7	20.78	4.547%	13.365%
4	ComputeScalar	const	4	6	31.00	4.744%	7.872%
5	NestedLoopJoin (var. outer side)	var	9	17	61.11	0.915%	1.836%
6	ClusteredIndexSeek	var	4	19	35.75	0.751%	1.396%
7	Sort	var	5	14	57.20	1.018%	1.195%
8	ComputeScalar	var	5	26	40.80	0.360%	1.055%
9	MergeJoin	var	6	21	95.00	0.609%	0.630%
10	ClusteredIndexScan	var	4	25	59.50	0.381%	0.464%
11	HybridHashJoin (var. probe side)	var	6	32	74.00	0.199%	0.282%
12	DistinctSort	var	5	33	76.60	0.180%	0.250%
13	HybridHashJoin (var. build side)	var	6	47	79.67	0.070%	0.180%
14	ClusteredIndexSeek	const	4	43	66.00	0.082%	0.152%
15	NestedLoopJoin (var. inner side)	var	6	46	72.17	0.070%	0.138%
16	StreamAggregate	var	5	54	86.00	0.048%	0.100%
17	ClusteredIndexScan	const	4	51	73.50	0.053%	0.096%
18	HybridHashJoin	const	4	57	81.75	0.040%	0.088%
19	SecondaryIndexScan	const	4	58	83.25	0.040%	0.084%
20	IndexNestedLoopJoin (var. inner side)	var	6	92	100.00	0.008%	0.036%
21	SecondaryIndexSeek	const	4	86	109.75	0.009%	0.020%
22	StreamAggregate	const	4	112	118.25	0.004%	0.008%
23	SecondaryIndexScan	var	4	124	125.5	0.000%	0.000%

Table 5.15: Feature importance grouped and aggregated by operator type sorted by sum of feature importance.

also supported the higher importance of features for `HybridHashJoin` operators with variable probe side. For both types of operators, the most important feature is the minimum-maximum ratio of input cardinality (v5 and v6) of the variable input, i. e., build or probe input. In general, Table 5.15 validates the findings of our benchmark analysis in Section 5.1.4. The predictions of the random forest model depend on features for index seek, nested-loop based join, and sort operators with variable input. All of these operators have been identified to bear a high risk of performance regression for varying input sizes.

5.4 Related Work

The problem of inaccurate selectivity estimates emerged with the first cost-based query optimizer. Since then, techniques for robust query processing are in the focus of database research. Different research directions have been explored over the years including methods to improve selectivity estimates, techniques to improve the choice of execution plans, and novel data processing algorithms, which are more tolerant with respect to estimation errors, and benchmarks to assess different robustness aspects of database systems.

In the following we discuss prior work from three areas of database research that are related to the contributions in this chapter, namely benchmarking DBMS robustness, query optimization for robust execution plans, and applications of machine learning approaches in database systems.

Robustness Benchmarks In Section 5.1, we presented the results of an experimental study to analyze performance variations of query execution plans and to identify plan features that cause plans to be sensitive to changing execution conditions.

Conventional database benchmarks focus on processing performance, i. e., transactional throughput or query execution performance. Wiener et al. [204] propose to extend benchmarks to include the assessment query execution robustness, i. e., the capability of a DBMS to handle unexpected execution conditions. The authors propose three robustness metrics, 1) *Optimality*: How much does the cost of an implementation (or plan) vary from the best known implementation (or plan)?, 2) *Consistency*: How does the performance of an implementation (or plan) vary across different execution conditions?, and 3) *Graceful degradation*: How much impact have small changes of execution conditions on the performance of an implementation (or plan)? Consistency is measured as the variance of execution time across different execution conditions. This definition is very close to our definition of plan robustness, i. e., the coefficient of variation which is a normalized variation metric. In addition, the authors proposed techniques to visualize the robustness of operators and plans [100]. Both papers present results of rather confined benchmarks to assess the robustness of table access techniques and methods to obtain a sorted result. In contrast, we have conducted a more extensive performance study to measure the execution time of complex analytical queries across a wide range of input sizes and memory settings.

The Picasso database query optimizer visualizer [179] is a tool to display the plan space of query optimizers. For a given query with one or two parameterized predicates, Picasso calls the optimizer of a DBMS with varying parameter bindings over the full range of predicate selectivities and retains the returned query execution plans. Picasso plots the optimizer's plan choices of the whole selectivity space by representing each distinct plan by a unique color. The paper shows several plan space plots generated for modified TPC-H queries, which are optimized by different database optimizers. These plots show that optimizer plan choices are often sensible for small

changes of cardinality estimates. Since cardinalities are notoriously hard to estimate and often off by large factors, the complexity of the plan space becomes questionable. In addition, Picasso generates three-dimensional plots of the estimated costs of a plan for varying selectivities. In our work, we follow a similar approach as Picasso. In fact, we use Picasso's TPC-H query templates with parameterized predicates and also call an optimizer with varying parameter bindings to collect different query execution plans. However, we actually execute these plans over a range of varying selectivity and memory settings, measure their execution times, and analyze their behavior for different execution conditions.

Waas et al. propose a method to measure the quality of optimizer cost models [202]. The method is based on the comparison of the two plan rankings defined by the optimizer's estimated cost and the actual execution time for different plans of a given query. In a follow-up paper [105] the method is refined to enable comparisons of the accuracy and optimality of different query optimizers. Both methods rely on optimizer hints to enforce the optimization and execution of different plans for the same query. In our work, we use optimizer hints to completely specify the plans to be executed for varying execution conditions.

Finally, there are a few papers that analyze the performance of individual operators. For example, Schneider et al. investigate the performance of four join algorithms in a distributed shared-nothing database system [183]. Haas et al. present detailed cost formulas for several join methods and validate these against benchmarks across varying parameters, such as memory and buffer size [107].

Robust Query Processing Different techniques to improve the robustness of query processing have been proposed. In the following, we discuss some research directions and present notable approaches.

Inaccurate selectivity estimates are a root cause for unexpected query performance. Consequently, selectivity estimation has been in the focus of database research for a long time. The first methods for selectivity estimation have been proposed in the paper that presented cost-based query optimization [186]. These methods are still the basis for the cardinality estimators in many database systems. To improve selectivity estimates for local predicates several techniques, including histograms [130, 135, 176] and wavelets [160], have been proposed. Further techniques are based on statistics collection for query expressions [39, 75], methods to avoid the independence assumption [158, 177], selectivity estimation from samples [116, 154], and approaches to leverage query feedback to improve selectivity estimates [192].

Parametric query optimization is an approach to improve plan choices for queries with parameters. Multiple plans for different parameter bindings of a query are precomputed. When a query with a parameter binding is received, the best plan for the binding is selected from the precomputed set and executed [31, 132]. In a recent work, Chaudhuri et al. generate plans for several parameter bindings, cross-cost these plans for all other bindings, and choose the plan with the

least variance in estimated plan cost [50]. As our performance study indicates, estimating the variance of plan execution time using optimizer estimates might not yield accurate results. Instead, our machine-learned regression model improved the accuracy of plan risk predictions. Another recent approach computes the set of plans which are optimal for some area of the parameter selectivity range and cover the full selectivity space [74]. This technique does not rely on selectivity estimation at all. Instead, the right plan is identified by partially executing plans from the set.

Other approaches try to model the uncertainty of selectivity estimates as probability distributions and incorporate this information into the query optimization process. Chu et al. propose least expected cost optimization [55, 56], which tracks histograms of cardinality and cost estimates of the probability distribution along the operators of plans at optimization time. Finally, the plan with the lowest expected cost, i. e., the plan that minimizes the execution cost over the probability distribution of the uncertain selectivities is chosen. Babcock and Chaudhuri propose a tuning knob to trade execution performance for predictable execution times [17]. The selectivity estimate that is used for query optimization is taken from a specified position of the cumulative probability distribution. In Section 5.2.2, we describe how the approach of Babcock and Chaudhuri relates to our approach of risk-weighted plan costing and conduct a brief experimental comparison.

Several approaches to improve plan robustness at optimization time have been proposed, which are based on the results of the Picasso project. Harish et al. reduce plan space diagrams by replacing plans by a neighboring plan that has similar cost but a more robust behavior for different input sizes [59, 60]. While this is an offline approach, Abhirama et al. propose a dynamic-programming-based online approach, which considers robust plan choices during plan enumeration. Both approaches use plan cross-costing, to obtain cost estimates for (sub)plans for specific points in the selectivity space.

The aforementioned approaches mainly focus on choosing a good execution plan at compile time. Another area of research has been adaptive query processing, i. e., techniques that are able to adjust or mitigate wrong optimizer decisions during query execution. Eddies [16] are a query processing technique that continuously reorders join operators while a query is executed. The Generalized Join [98] aims to unite the beneficial behaviors of Hash, Sort-Merge, and Index-Nested-Loop Joins with the goal to mitigate the optimization problem of choosing the right join algorithm. Deshpande et al. [66] present an extensive overview of further work in the direction of adaptive query processing.

Finally, there are several hybrid approaches that combine uncertainty-aware optimization and runtime adaption. Graefe and Ward [94] inject choose-plan operators into query plans to switch the execution strategies at runtime if certain conditions are met. Other approaches add check operators to collect statistics and compare them to a validity range specified by the optimizer.

If thresholds are exceeded, a check operator triggers a reconfiguration of the current execution plan [138], a switch to a previously computed plan alternative [19], or a (complete) re-optimization of the query leveraging the collected statistical information [159].

ML predictions in DBMS The use of machine learning techniques to predict the execution time, progress, or resource consumption of query execution plans is a fairly new trend in the database research community.

Ganapathi et al. present a machine learning approach to predict various performance characteristics of a plan, such as disk I/O, record, and message count, before the plan is executed [88]. Queries are represented as feature vectors and mapped into a similarity space using a Kernel Canonical Correlation Analysis (KCCA). To predict the performance characteristics of a plan, the three closest plans from the training set are fetched from the similarity space and the average of their performance characteristics is returned. The feature vector is constructed by counting and summing the input cardinalities of each operator type.

Another approach to predict the resource consumption (CPU time and logical reads) of query execution plans is based on hybrid prediction models of boosted regression trees and a scaling method for each operator type [150]. Since the range of prediction values of regression trees is limited to the value range observed during training, a scaling function is used to model the asymptotic behavior of the operators. The resource consumption of a plan is predicted by summing the predictions of its individual operators. While this approach leads to improved prediction performance and better generalization for previously unseen queries, it is not applicable to our prediction problem because it is not trivial to compute a plan's risk score given the risk scores of its individual operators.

Akdere et al. train linear regression or SVM models to predict the performance of plans before they are executed [8]. Plans are modeled with both coarse-grain plan-level features and fine-grain operator-level features. It is shown that a hybrid of both approaches yields the best results.

König et al. propose a method for progress estimation of plans before and during their execution [142]. Instead of directly predicting the progress, the authors follow a meta-prediction approach and use statistical models to select an estimator from a set of previously proposed progress estimation methods. For each estimator they train a model of boosted regression trees to predict the estimation error of plans. The feature vectors of this approach are based on static compile time features as well as runtime features which are observed during execution. Our approach is also based on regression trees, however extremely randomized trees performed better than boosted regression trees for our learning problem.

Our approach of predicting risk scores for query execution plans uses operator type aggregated features similar to Ganapathi et al.'s approach [88] and random forest regression models as Li

et al. [150] and König et al. [142]. We are not aware of any other work that attempts to predict the robustness or riskiness of query execution plans using machine learning techniques.

5.5 Summary

Query execution plans differ in their sensitivity for varying execution conditions. While some plans have nearly constant execution time if the size of a certain input increases by an order of magnitude, other plans suffer from a significant regression. In this chapter, we discussed and analyzed the differences in sensitivity of query plans for changing execution conditions.

In order to evaluate the sensitivity of query execution plans, we conducted an extensive experimental study and measured the execution time of operators, join trees, and complex analytical queries for varying execution conditions. We analyzed the execution behavior and identified plan patterns that are able to cause significant variations in execution time for changing input sizes and memory budgets. Our analysis also showed that it is not possible to assess the risk of a plan by looking at individual operators only. Instead, a plan as a whole needs to be taken into account in order to reason about its sensitivity with respect to varying execution conditions.

We proposed a risk score to assess the sensitivity of query execution plans for varying execution conditions, such as predicate parameter bindings and main memory budgets. The risk score is based on the coefficient of variation of a plan's execution time for changing execution conditions. We show how the risk score can be used to prevent risky plan choices by computing a risk-aware plan cost metric and evaluate this method using the plan execution data from our previous experiments.

Subsequently, we proposed a method to predict the risk score of query execution plans using a machine-learned regression model. We discussed the design of the feature vector to capture the aspects of a query plan, which are relevant to predict its risk. The evaluation of a random forest regression model which was trained using our plan execution data showed that its prediction performance is significantly better than risk predictions computed from optimizer estimates. We found that inaccurate predictions of the regression model can be accounted to insufficient training data. This is a problem especially for rare high-risk patterns. An analysis of the importance of our features seconds the findings of the manual analysis of the plan execution data. Operators that we deemed risky provided important features for the machine-learned regression model.

6 Conclusion and Outlook

Today, data analysis is pervasive in many domains. Growing data set sizes and more compute-intensive analysis methods demand for parallel data processing. In this thesis, we presented our work to improve data analysis by easing the specification and increasing the performance and efficiency of parallel analytical data flows. In the following, we conclude this thesis, discuss open problems, and propose future research directions.

In Chapter 3 we proposed the PACT programming model. It combines features of declarative program specification, concepts of parallel programming models, and treats user-defined functions (UDF) and data types as first class citizens. This combination results in an expressive programming model that includes several benefits of declarative program specification. Declarativity decouples the specification of a program is from its execution. Consequently, the execution of a program can be optimized with respect to the program's input and execution environment. Furthermore, declarative concepts ease the definition of programs because the user can focus on the semantics of a program while the optimizer takes care of the actual execution strategies. PACT programs are optimized similar to SQL queries in relational database systems and execute user-defined functions in parallel similar to MapReduce programs.

Chapter 4 presented techniques to enhance the optimization capabilities for data flow programs with user-defined functions, such as PACT programs. Optimizing these programs is challenging because their semantics are hidden in user-defined functions. Our approach uses static code analysis techniques to extract a few properties from user-defined functions. Based on these properties, we presented and proved sufficient conditions to decide whether two UDF operators can be reordered. The evaluation of our approach showed that important optimizations of relational database optimizers are supported by our approach including filter and projection push-down and join order optimization. These optimizations can yield significant performance improvements while being fully transparent to the user due to static code analysis.

Chapter 5 addressed a problem that is common for all cost-based optimization approaches, namely inaccurate optimizer estimates. While this is a hard problem in traditional relational database systems, it is certainly even more challenging in less controlled setups, such as general-purpose data analysis engines since less information about the data, the execution environment, and the executed programs is available. Imprecise estimates are challenging for cost-based optimizers because execution plans differ in their sensitivity with respect to imprecise estimates and changing execution conditions. In Chapter 5, we analyzed this sensitivity by conducting an extensive experimental study and identified features of query plans that can cause significantly

varying performance for changing execution conditions. We proposed a plan risk score to identify sensitive plans and showed how it can be used to prevent the choice of risky plans by query optimizers. Finally, we presented an approach to predict the risk score of query plans using a machine-learned regression model.

Chapters 3 and 4 focused on enabling the optimization of parallel data flows. Due to the declarative characteristics of PACT operators and the presented conditions to reorder operators with user-defined functions, programs specified in the PACT programming model can be optimized and translated into many different execution plans. We proposed cost-based optimization to identify the best plan from the plan space. However, the effectiveness of cost-based optimization depends on the availability of reliable cost estimates for plan alternatives.

This thesis did not discuss in detail how to compute cost estimates for parallel data flows that process in-situ data and consist of operators with user-defined functions. The traditional approach is based on cost functions for operators and cardinality estimates for all intermediate results. Both, reliable cardinality estimates as well as an accurate cost model, are usually not available due to missing base statistics and unknown operator semantics. In Chapter 5, we analyzed the effect of missing optimizer estimates and discussed approaches to reduce their impact on the quality of plan selection. However, also these approaches benefit from additional information and should only be considered as a safety net to prevent the selection of highly risky plans.

One direction for future research are reliable plan choices for massively parallel data flow programs. A possible approach could be incremental optimization and execution in combination with on-line statistics collection. Programs are partially planned and executed in a step-wise fashion. During the execution of a step, the processing engine collects statistics that help to plan the next step. This approach has a few interesting trade-offs that might be worth exploring. For instance, the statistics that can be collected depend on the choice of the currently executed step-plan and the statistics that will be helpful depend on the not-yet executed portion of the program. The size of a step is also an interesting choice, i. e., how much uncertainty (or risk) does the optimizer allow within a step.

Another area of future research is extending the optimization of programs with user-defined functions. One option are more elaborate code analysis techniques to (partially) extract the semantics of user-defined functions. This information might facilitate more optimizations or help estimating the cost or selectivity of a UDF. Other approaches could aim to automatically improve the code of a user-defined function or even fuse it with other UDFs or engine code.

Bibliography

[1] Daniel J. Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. The beckman report on database research. *SIGMOD Record*, 43(3):61–70, 2014.

[2] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. Stability-conscious query optimization. 2009.

[3] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[4] Apache accumulo. URL: http://accumulo.apache.org.

[5] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, 2000.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Pearson, 2006.

[7] Anastasia Ailamaki, Verena Kantere, and Debabrata Dash. Managing scientific data. *Commun. ACM*, 53(6):68–78, 2010.

[8] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. Learning-based query performance modeling and prediction. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 390–401. IEEE, 2012.

[9] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively parallel data analysis with pacts on nephele. *PVLDB*, 3(2):1625–1628, 2010.

[10] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, pages 1–26, 2014.

[11] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Mapreduce and pact - comparing data parallel programming models. In *BTW*, pages 25–44, 2011.

[12] Peter Alvaro, Neil Conway, and Andrew Krioukov. Multi-query optimization for parallel dataflow systems. URL: http://neilconway.org/docs/286_mqo.pdf, May 2009.

[13] Apache ambari. URL: http://ambari.apache.org.

[14] D.H. Brown Associates. Db2 udb vs. oracle8i: Total cost of ownership. 2000.

[15] Asterixdb. URL: http://asterix.ics.uci.edu.

[16] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM.

[17] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD Conference*, pages 119–130, 2005.

[18] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *SoCC*, pages 137–142, 2010.

[19] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive re-optimization. In *SIGMOD Conference*, pages 107–118, 2005.

[20] Monya Baker. Next-generation sequencing: adjusting to data overload. *nature methods*, 7(7):495–499, 2010.

[21] Roger Bamford, D Butler, Boris Klots, and N MacNaughton. Architecture of oracle parallel server. In *VLDB*, volume 98, pages 669–670, 1998.

[22] Chaitanya K. Baru, Gilles Fecteau, Ambuj Goyal, H Hsiao, Anant Jhingran, Sriram Padmanabhan, George P. Copeland, and Walter G. Wilson. Db2 parallel edition. *IBM Systems journal*, 34(2):292–322, 1995.

[23] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *SoCC '10: Proceedings of the ACM Symposium on Cloud Computing 2010*, pages 119–130, New York, NY, USA, 2010. ACM.

[24] Jacek Becla, Andrew Hanushevsky, Sergei Nikolaev, Ghaleb Abdulla, Alexander S. Szalay, María A. Nieto-Santisteban, Ani Thakar, and Jim Gray. Designing a multi-petabyte database for lsst. *CoRR*, abs/cs/0604112, 2006.

[25] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[26] Björn Bergsten, Michel Couprie, and Patrick Valduriez. Prototyping dbs3, a shared-memory parallel database system. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida, December 4-6, 1991*, pages 226–234, 1991.

[27] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

[28] Kevin Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 2011.

[29] Apache bigtop. URL: http://bigtop.apache.org.

[30] Infosphere biginsights. URL: http://www-01.ibm.com/software/data/infosphere/biginsights/.

[31] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. Progressive parametric query optimization. *IEEE Trans. Knowl. Data Eng.*, 21(4):582–594, 2009.

[32] José A. Blakeley, Paul A. Dyke, César Galindo-Legaria, Nicole James, Christian Kleinerman, Matt Peebles, Richard Tkachuk, and Vaughn Washington. Microsoft sql server parallel data warehouse: Architecture overview. In *Enabling Real-Time Business Intelligence*, pages 53–64. Springer, 2012.

[33] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD Conference*, pages 975–986, 2010.

[34] Christoph Boden, Marcel Karnstedt, Miriam Fernández, and Volker Markl. Large-scale social-media analytics on stratosphere. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 257–260, 2013.

[35] Haran Boral, William Alexander, Larry Clay, George P. Copeland, Scott Danforth, Michael J. Franklin, Brian E. Hart, Marc G. Smith, and Patrick Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. Knowl. Data Eng.*, 2(1):4–24, 1990.

[36] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[37] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[38] Nicolas Bruno, Sameer Agarwal, Srikanth Kandula, Bing Shi, Ming-Chuan Wu, and Jingren Zhou. Recurring job optimization in scope. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 805–806, 2012.

[39] Nicolas Bruno and Surajit Chaudhuri. Efficient creation of statistics over query expressions. In *ICDE*, pages 201–212, 2003.

[40] Nicolas Bruno, Sapna Jain, and Jingren Zhou. Continuous cloud-scale query optimization and processing. *PVLDB*, 6(11):961–972, 2013.

[41] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[42] Michael J. Cafarella and Christopher Ré. Manimal: Relational optimization for data-intensive programs. In *WebDB*, 2010.

[43] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Paxquery: Efficient parallel processing of complex xquery. *IEEE Trans. Knowl. Data Eng.*, 27(7):1977–1991, 2015.

[44] Cascading. URL: http://www.cascading.org/.

[45] Cern Computing. URL: http://home.cern/about/computing.

[46] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM*, 10(10-17):30, 2010.

[47] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

[48] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Tushar Chandra Mike Burrows, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[49] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. Tenzing a sql implementation on the mapreduce framework. In *Proceedings of VLDB*, pages 1318–1327, 2011.

[50] Surajit Chaudhuri, Hongrae Lee, and Vivek R. Narasayya. Variance aware optimization of parameterized queries. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 531–542, New York, NY, USA, 2010. ACM.

[51] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *VLDB*, pages 354–366, 1994.

[52] Surajit Chaudhuri and Kyuseok Shim. An overview of cost-based optimization of queries with aggregates. *IEEE Data Eng. Bull.*, 18(3):3–9, 1995.

[53] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[54] Ming-Syan Chen and Philip S. Yu. Interleaving a join sequence with semijoins in distributed query processing. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):611–621, 1992.

[55] Francis C. Chu, Joseph Y. Halpern, and Johannes Gehrke. Least expected cost query optimization: What can we expect? In *PODS*, pages 293–302, 2002.

[56] Francis C. Chu, Joseph Y. Halpern, and Praveen Seshadri. Least expected cost query optimization: An exercise in utility. In *PODS*, pages 138–147, 1999.

[57] Jonathan Cohen. Graph Twiddling in a MapReduce World. *Computing in Science and Engineering*, 11:29–41, 2009.

[58] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *NSDI*, pages 313–328, 2010.

[59] Harish D, Pooja N. Darera, and Jayant R. Haritsa. On the production of anorexic plan diagrams. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1081–1092. VLDB Endowment, 2007.

[60] Harish D., Pooja N. Darera, and Jayant R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.

[61] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[62] George S Davidson, Jim R Cowie, Stephen C Helmreich, Ron A Zacharski, and Kevin W Boyack. *Data-centric computing with the netezza architecture*. United States. Department of Energy, 2006.

[63] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[64] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[65] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[66] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Found. Trends databases*, 1(1):1–140, January 2007.

[67] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[68] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[69] David J DeWitt, Alan Halverson, Rimma Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. Split query processing in polybase. In *Proceedings of the 2013 international conference on Management of data*, pages 1255–1266. ACM, 2013.

[70] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[71] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[72] Dryad - Microsoft Research. URL: http://research.microsoft.com/projects/Dryad.

[73] DryadLINQ - Microsoft Research. URL: http://research.microsoft.com/projects/DryadLINQ.

[74] Anshuman Dutt and Jayant R. Haritsa. Plan bouquets: query processing without selectivity estimation. In *SIGMOD Conference*, pages 1039–1050, 2014.

[75] Amr El-Helw, Ihab F Ilyas, and Calisto Zuzarte. Statadvisor: recommending statistical views. *Proceedings of the VLDB Endowment*, 2(2):1306–1317, 2009.

[76] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011.

[77] Amazon Elastic MapReduce. URL: https://aws.amazon.com/elasticmapreduce.

[78] Susanne Englert, Jim Gray, Terrye Kocher, and Praful Shah. A benchmark of nonstop sql release 2 demonstrating near-linear speedup and scaleup on large databases. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):245–246, 1990.

[79] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas, Daniel Warneke, and Volker Markl. Iterative parallel data processing with stratosphere: an inside look. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1053–1056, 2013.

[80] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning fast iterative data flows. *PVLDB*, 5(11):1268–1279, 2012.

[81] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An optimization framework for map-reduce queries. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 26–37, 2012.

[82] Pit Fender and Guido Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *ICDE*, pages 864–875, 2011.

[83] Pit Fender, Guido Moerkotte, Thomas Neumann, and Viktor Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*, pages 414–425, 2012.

[84] Apache Flink. URL: http://flink.apache.org.

[85] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7):419–429, 2011.

[86] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.

[87] Eric Friedman, Peter M. Pawlowski, and John Cieslewicz. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB*, 2(2):1402–1413, 2009.

[88] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael I Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.

[89] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *SIGMOD Conference*, pages 9–18, 1992.

[90] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a highlevel dataflow system on top of mapreduce: The pig experience. *PVLDB*, 2(2):1414–1425, 2009.

[91] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[92] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[93] Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 231–242, 2011.

[94] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 358–366, New York, NY, USA, 1989. ACM.

[95] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 102–111, 1990.

[96] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[97] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[98] Goetz Graefe. A generalized join algorithm. In *BTW*, pages 267–286, 2011.

[99] Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. Robust query processing (dagstuhl seminar 12321). *Dagstuhl Reports*, 2(8):1–15, 2012.

[100] Goetz Graefe, Harumi A. Kuno, and Janet L. Wiener. Visualizing the robustness of query execution. In *CIDR*, 2009.

[101] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218, 1993.

[102] Götz Graefe, Arnd Christian König, Harumi Anne Kuno, Volker Markl, and Kai-Uwe Sattler. 10381 Summary and Abstracts Collection – Robust Query Processing. In Goetz Graefe, Arnd Christian König, Harumi Anne Kuno, Volker Markl, and Kai-Uwe Sattler, editors, *Robust Query Processing*, number 10381 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[103] Pivotal Greenplum Database. URL: http://www.pivotal.io/big-data/pivotal-greenplum-database.

[104] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *SIGMOD Conference*, pages 1063–1066, 2009.

[105] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. Testing the accuracy of query optimizers. In *DBTest*, page 11, 2012.

[106] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 121–133, 2012.

[107] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.

[108] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In *SIGMOD Conference*, pages 377–388, 1989.

[109] Apache Hadoop. URL: http://hadoop.apache.org.

[110] Gary Hallmark. Oracle parallel warehouse server. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 314–320. IEEE, 1997.

[111] Apache Hama. URL: http://hama.apache.org.

[112] Li Haoyuan, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SoCC '14: Proceedings of the ACM Symposium on Cloud Computing 2014*, New York, NY, USA, 2014. ACM.

[113] Waqar Hasan. *Optimization of SQL Queries for Parallel Machines*, volume 1182 of *Lecture Notes in Computer Science*. Springer, 1996.

[114] Waqar Hasan, Daniela Florescu, and Patrick Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 25(3):28–33, 1996.

[115] Apache HBase. URL: http://hbase.apache.org.

[116] Max Heimel and Volker Markl. A first step towards gpu-assisted query optimization. In *ADMS@VLDB*, pages 33–44, 2012.

[117] Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. Meteor/sopremo: An extensible query language and operator model. *Workshop on End-to-end Management of Big Data, Istanbul, Turkey*, 2012.

[118] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[119] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, pages 267–276, 1993.

[120] Herodotos Herodotou. Hadoop performance models. *CoRR*, abs/1106.0940, 2011.

[121] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4, 2011.

[122] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, pages 261–272, 2011.

[123] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[124] Apache Hive. URL: http://hive.apache.org.

[125] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.

[126] Fabian Hueske, Aljoscha Krettek, and Kostas Tzoumas. Enabling operator reordering in data flow programs through static code analysis. In *XLDI Workshop, affiliated with ICFP*, 2012.

[127] Fabian Hueske and Volker Markl. Optimization of massively parallel data flows. In *Large-Scale Data Analytics*, pages 41–74. Springer New York, 2014.

[128] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. Peeking into the optimization of data flow programs with mapreduce-style udfs. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1292–1295, 2013.

[129] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[130] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 19–30. VLDB Endowment, 2003.

[131] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD Conference*, pages 268–277, 1991.

[132] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. *VLDB J.*, 6(2):132–151, 1997.

[133] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[134] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 987–994, 2009.

[135] HV Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *VLDB*, volume 98, pages 275–286, 1998.

[136] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *PVLDB*, 4(6):385–396, 2011.

[137] Anant Jhingran, Timothy Malkemus, and Sriram Padmanabhan. Query optimization in DB2 parallel edition. *IEEE Data Eng. Bull.*, 20(2):27–34, 1997.

[138] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD Conference*, pages 106–117, 1998.

[139] Vasiliki Kalavri, Stephan Ewen, Kostas Tzoumas, Vladimir Vlassov, Volker Markl, and Seif Haridi. Asymmetry in large-scale graph analysis, explained. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, pages 4:1–4:7, 2014.

[140] Vasiliki Kalavri, Vladimir Vlassov, and Per Brand. Ponic: Using stratosphere to speed up pig analytics. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 279–290, 2013.

[141] Qifa Ke, Michael Isard, and Yuan Yu. Optimus: a dynamic rewriting framework for data-parallel execution plans. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, pages 15–28, 2013.

[142] Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A statistical approach towards robust progress estimation. *Proceedings of the VLDB Endowment*, 5(4):382–393, 2011.

[143] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[144] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[145] Alexandros Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *PVLDB*, 5(12):2032–2033, 2012.

[146] Rosana S. G. Lanzelotte, Patrick Valduriez, Mohamed Zaït, and Mikal Ziane. Invited project review: Industrial-strength parallel query optimization: issues and lessons. *Inf. Syst.*, 19(4):311–330, 1994.

[147] Mong-Li Lee, Masaru Kitsuregawa, Beng Chin Ooi, Kian-Lee Tan, and Anirban Mondal. Towards self-tuning data placement in parallel database systems. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 225–236, 2000.

[148] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[149] Marcus Leich, Jochen Adamek, Moritz Schubotz, Arvid Heise, Astrid Rheinländer, and Volker Markl. Applying stratosphere for big data analytics. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings*, pages 507–510, 2013.

[150] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. *Proceedings of the VLDB Endowment*, 5(11):1555–1566, 2012.

[151] Harold Lim, Herodotos Herodotou, and Shivnath Babu. Stubby: A transformation-based optimizer for mapreduce workflows. *PVLDB*, 5(11):1196–1207, 2012.

[152] Jimmy Lin and Alek Kolcz. Large-scale machine learning at twitter. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 793–804. ACM, 2012.

[153] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *SIGMOD Conference*, pages 961–972, 2011.

[154] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 1–11, New York, NY, USA, 1990. ACM.

[155] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[156] Apache Mahout. URL: http://mahout.apache.org.

[157] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[158] Volker Markl, Nimrod Megiddo, Marcel Kutsch, Tam Minh Tran, Peter J. Haas, and Utkarsh Srivastava. Consistently estimating the selectivity of conjuncts of predicates. In *VLDB*, pages 373–384, 2005.

[159] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In *SIGMOD Conference*, pages 659–670, 2004.

[160] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. *SIGMOD Record*, 27(2):448–459, 1998.

[161] Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB J.*, 6(1):53–72, 1997.

[162] Apache Mesos. URL: http://mesos.apache.org.

[163] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *SIGMOD Conference*, pages 539–552, 2008.

[164] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.

[165] Apache MRQL. URL: http://mrql.incubator.apache.org.

[166] Clara Nippl and Bernhard Mitschang. Topaz: a cost-based, rule-driven, multi-phase parallelizer. In *VLDB*, pages 251–262, 1998.

[167] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 3(1):494–505, 2010.

[168] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[169] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[170] Apache Oozie. URL: http://oozie.apache.org.

[171] M. Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. In *Computing Handbook, Third Edition: Information Systems and Information Technology*, pages 13: 1–24. 2014.

[172] Apache Parquet (incubating). URL: http://parquet.incubator.apache.org.

[173] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[174] Picasso website. URL: http://dsl.serc.iisc.ernet.in/projects/PICASSO/.

[175] Apache Pig. URL: http://pig.apache.org.

[176] Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD Record*, 25(2):294–305, 1996.

[177] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, pages 486–495, 1997.

[178] Erhard Rahm. Parallel query processing in shared disk database systems. *SIGMOD Record*, 22(4):32–37, 1993.

[179] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *VLDB*, pages 1228–1240, 2005.

[180] Astrid Rheinländer, Arvid Heise, Fabian Hueske, Ulf Leser, and Felix Naumann. SOFA: an extensible logical optimizer for udf-heavy dataflows. *CoRR*, abs/1311.6335, 2013.

[181] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *VLDB J.*, 23(3):469–494, 2014.

[182] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "all roads lead to rome": optimistic recovery for distributed iterative data processing. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 1919–1928, 2013.

[183] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989.*, pages 110–121, 1989.

[184] scikit-learn website. URL: http://scikit-learn.org.

[185] SciPy website. URL: http://scipy.org.

[186] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34, 1979.

[187] Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David DeWitt, and César Galindo-Legaria. Query optimization in microsoft sql server pdw. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 767–776. ACM, 2012.

[188] SNAP website. URL: http://snap.stanford.edu/data/index.html.

[189] Soot: a Java Optimization Framework. URL: http://www.sable.mcgill.ca/soot/.

[190] Apache Spark. URL: http://spark.apache.org.

[191] Jaideep Srivastava and Gary Elsesser. Optimizing multi-join queries in parallel relational databases. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS 1993), Issues, Architectures, and Algorithms, San Diego, CA, USA, January 20-23, 1993*, pages 84–92, 1993.

[192] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. Leo - db2's learning optimizer. In *VLDB*, pages 19–28, 2001.

[193] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[194] Michael Stonebraker, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[195] Stratosphere. URL: http://stratosphere.eu.

[196] Tachyon. URL: http://tachyon-project.org.

[197] Apache Tez. URL: http://tez.apache.org.

[198] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang 0002, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop. In *ICDE*, pages 996–1005, 2010.

[199] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[200] TPC-H website. URL: http://www.tpc.org/tpch/.

[201] Florian M. Waas. Beyond conventional data warehousing - massively parallel data processing with greenplum database. In *Informal Proceedings of the Second International Workshop on Business Intelligence for the Real-Time Enterprise, BIRTE 2008, in conjunction with VLDB'08, August 24, 2008, Auckland, New Zealand*, 2008.

[202] Florian M. Waas, Leo Giakoumakis, and Shin Zhang. Plan space analysis: an early warning system to detect plan regressions in cost-based optimizers. In *DBTest*, page 2, 2011.

[203] Daniel Warneke and Odej Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *SC-MTAGS*, 2009.

[204] Janet L. Wiener, Harumi A. Kuno, and Goetz Graefe. Benchmarking query execution robustness. In *TPCTC*, pages 153–166, 2009.

[205] Apache Hadoop - YARN. URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[206] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.

[207] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[208] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010.

[209] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, pages 29–42, 2008.

[210] Hansjorg Zeller. Parallel query execution in nonstop sql. In *Compcon Spring'90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, pages 484–487. IEEE, 1990.

[211] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y. Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 295–308, 2012.

[212] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. Scope: parallel databases meet mapreduce. *VLDB J.*, 21(5):611–636, 2012.

[213] Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071, 2010.