ROBUST AND ADAPTIVE QUERY PROCESSING IN HYBRID TRANSACTIONAL/ANALYTICAL DATABASE SYSTEMS

Zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)
der

FAKULTÄT FÜR INFORMATIK
UND AUTOMATISIERUNG
der
TECHNISCHEN UNIVERSITÄT ILMENAU
vorgelegte Dissertation von

Florian Wolf
geboren am 28.01.1990 in Weimar

TECHNISCHE UNIVERSITÄT
ILMENAU

Tag der Einreichung: 27.09.2018

Erster Gutachter:	Univ.-Prof. Dr.-Ing. habil. Kai-Uwe Sattler
Zweiter Gutachter:	Univ.-Prof. Dr.-Ing. Wolfgang Lehner
Dritter Gutachter:	Univ.-Prof. Dr. Thomas Neumann

Tag des öffentlichen Teils der wissenschaftlichen Aussprache: 18.12.2018

urn:nbn:de:gbv:ilm1-2018000576

Dedicated to my family.

ABSTRACT

The quality of query execution plans in database systems determines how fast a query can be processed. Conventional query optimization may still select sub-optimal or even bad query execution plans, due to errors in the cardinality estimation. In this work, we address limitations and unsolved problems of Robust and Adaptive Query Processing, with the goal of improving the detection and compensation of sub-optimal query execution plans. We demonstrate that existing heuristics cannot sufficiently characterize the intermediate result cardinalities, for which a given query execution plan remains optimal, and present an algorithm to calculate precise optimality ranges. The compensation of sub-optimal query execution plans is a complementary problem. We describe metrics to quantify the robustness of query execution plans with respect to cardinality estimations errors. In queries with cardinality estimation errors, our corresponding robust plan selection strategy chooses query execution plans, which are up to 3.49× faster, compared to the estimated cheapest plans. Furthermore, we present an adaptive query processor to compensate sub-optimal query execution plans. It collects true cardinalities of intermediate results at query execution time to re-optimize the currently running query. We show that the overall effort for re-optimizations and plan switches is similar to the initial optimization. Our adaptive query processor can execute queries up to 5.19× faster, compared to a conventional query processor.

ZUSAMMENFASSUNG

Die Qualität von Anfrageausführungsplänen in Datenbank Systemen bestimmt, wie schnell eine Anfrage verarbeitet werden kann. Aufgrund von Fehlern in der Kardinalitätsschätzung können konventionelle Anfrageoptimierer immer noch sub-optimale oder sogar schlechte Anfrageausführungsplänen auswählen. In dieser Arbeit behandeln wir Einschränkungen und ungelöste Probleme robuster und adaptiver Anfrageverarbeitung, um die Erkennung und den Ausgleich sub-optimaler Anfrageausführungspläne zu verbessern. Wir zeigen, dass bestehende Heuristiken nicht entscheiden können, für welche Kardinalitäten ein Anfrageausführungsplan optimal ist, und stellen einen Algorithmus vor, der präzise Optimalitätsbereiche berechnen kann. Der Ausgleich von sub-optimalen Anfrageausführungsplänen ist ein ergänzendes Problem. Wir beschreiben Metriken, welche die

Robustheit von Anfrageausführungsplänen gegenüber Fehlern in der Kardinalitätsschätzung quantifizieren können. Unsere robuste Planauswahlstrategie, die auf Robustheitsmetriken aufbaut, kann Pläne finden, die bei Fehlern in der Kardinalitätsschätzung bis zu 3.49× schneller sind als die geschätzt günstigsten Pläne. Des Weiteren stellen wir einen adaptiven Anfrageverarbeiter vor, der sub-optimale Anfrageausführungspläne ausgleichen kann. Er erfasst die wahren Kardinalitäten von Zwischenergebnissen während der Anfrageausführung, um damit die aktuell laufende Anfrage zu re-optimieren. Wir zeigen, dass der gesamte Aufwand für Re-Optimierungen und Planänderungen einer initialen Optimierung entspricht. Unser adaptiver Anfrageverarbeiter kann Anfragen bis zu 5.19× schneller ausführen als ein konventioneller Anfrageverarbeiter.

PUBLICATIONS

[1] Florian Wolf, Heiko Betz, Francis Gropengießer, and Kai-Uwe Sattler. "Hibernating in the Cloud – Implementation and Evaluation of Object-NoSQL-Mapping." In: *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)*. BTW '13. Magdeburg, Germany: Gesellschaft für Informatik, 2013.

[2] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. "Scaling up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling." In: *6th TPC Technology Conference on Performance Evaluation and Benchmarking*. TPCTC '14. Hangzhou, China: Springer International Publishing, 2014.

[3] Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, and Kai-Uwe Sattler. "Extending Database Task Schedulers for Multi-threaded Application Code." In: *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. SSDBM '15. La Jolla, CA, USA: ACM, 2015.

[4] Florian Wolf, Norman May, Paul R. Willems, and Kai-Uwe Sattler. "On the Calculation of Optimality Ranges for Relational Query Execution Plans." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018.

[5] Florian Wolf, Michael Brendle, Norman May, Paul R. Willems, Kai-Uwe Sattler, and Michael Grossniklaus. "Robustness Metrics for Relational Query Execution Plans." In: *Proceedings of the VLDB Endowment 11.11*. VLDB '18. Rio de Janeiro, Brazil: VLDB Endowment, 2018.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Prof. Kai-Uwe Sattler for being a great teacher at the TU Ilmenau, who supported me on my journey from a bachelor and master student, to the final submission of my dissertation. I am truly grateful for an excellent database education, a bachelor thesis topic that resulted in my first paper, and the support during my master studies. In the last three and a half years as a PhD candidate, I always enjoyed our open and fair collaboration.

When Prof. Kai-Uwe Sattler introduced me at SAP, it was the beginning of a great time in Walldorf. I would like to thank Dr. Norman May for supervising me during my internship, my master thesis project, and my PhD project in the SAP HANA group. I am certainly thankful for his effort and commitment, and always appreciated the open and honest discussions we had. In addition, I would like to thank Dr. Paul R. Willems for co-supervising my PhD project at SAP. His expertise and understanding of complex problems was always an enrichment, and I acknowledge our professional collaboration.

Doing a PhD project in the SAP HANA group is a privilege, and I would like to thank SAP and the SAP HANA group for the funding. In particular, I would like to thank Arne Schwarz as a great mamanger for his valuable support during my time as a student and a PhD candidate at SAP. I furthermore want to express my thanks to Dr. Iraklis Psaroudakis for being a role model and supervisor in my first year at SAP. In addition, I would like to thank Prof. Wolfgang Lehner, Dr. Alexander Böhm, Daniel Schneiss, and Franz Färber for enabling the cutting-edge research in the SAP HANA group.

I was privileged to have great students that supported my research, namely Michael Brendle and Prakriti Bhardwaj. I would like to thank Michael Brendle for his effort and commitment during our work on the robustness metrics, and I wish him all the best for his PhD project.

The TU Ilmenau is an outstanding place dedicated to research and education. I am grateful for eight wonderful years. In particular, I would like to thank my colleagues at the Database and Information Systems group, Dr. Francis Gropengießer, Stephan Baumann, Felix Beier, Stefan Hagedorn, Philipp Götze, and Constantin Pohl.

Furthermore, I want to thank my colleagues in the SAP HANA campus, Dr. Iraklis Psaroudakis, Dr. David Kernert, Dr. Elena Vasilyeva, Dr. Robert Brunel, Dr. Ingo Müller, Dr. Marcus Paradies, Dr. Ismail Oukid, Dr. Hannes Rauhe, Max Wildemann, Radwan Deeb, Michael Rudolf, Stefan Noll, Lucas Lersch, Matthias Hauck, Georgios Psaropoulos, Robin Rehrmann, Thomas Bach, and Frank Tetzel for five fantastic years, and the constructive exchange on ideas and tech-

nologies. It has been a great honor to me, to work with such gifted people, who are blessed with intellect and perception.

Acknowledgments also go to the co-authors of my publications, namely Prof. Kai-Uwe Sattler, Prof. Thomas Neumann, Prof. Anastasia Ailamaki, Prof. Michael Grossniklaus, Dr. Norman May, Dr. Paul R. Willems, Dr. Iraklis Psaroudakis, Dr. Alexander Böhm, and Dr. Francis Gropengießer.

Next to Prof. Kai-Uwe Sattler, I would also like to thank Prof. Wolfgang Lehner and Prof. Thomas Neumann for reviewing my dissertation. I would like to thank Prof. Thomas Neumann for his inspirational talk on engineering high-performance database engines at the VLDB 2014 in China. It is incredible at which pace and quality Prof. Thomas Neumann and his group contribute to the database theory. His publications are an essential foundation for my work.

I am grateful for an amazing and inspiring summer school on data management in Dagstuhl in 2016, organized by Dr. Goetz Graefe. Furthermore, I would like to thank the reviewers of our papers for their valuable feedback, as well as the colleagues from Microsoft, Oracle, Amazon, IBM, and Tableau for their interest in our research.

Finally, I would like to thank my family and friends for their great support and backing. In particular, I would like to thank my parents for fostering my creativity, and encouraging me to pursue my goals. I owe you a lot. I would also like to thank my girlfriend for providing me with power and strength.

CONTENTS

1 INTRODUCTION

Declarative query languages are a key success factor for multiple kinds of database systems. Taking relational [1] or graph database systems [51] as an example, the declarative query languages SQL [2] and GCore [96] describe the result of a query instead of a procedure to calculate it. On the one hand, declarative query languages hide the complexity of finding the best procedure to calculate the query result. On the other hand, they delegate the complexity of finding the best procedure to the query optimizer of the database system.

Each query on a relational database is translated into logical relational algebra, which contains operators such as scans, projections, filters, aggregations and joins. Since there can be multiple implementations for logical relational operators, there is furthermore the physical relational algebra with physical relational operators. The goal of query optimization is to find a query execution plan in physical relational algebra, which achieves the minimal query execution time or resource consumption. To select a corresponding query execution plan, the query optimizer enumerates a set of query execution plan alternatives, which can grow exponentially with the number of joined tables. Next to the operator implementation, the performance of query execution plans depends on the execution order of operators. This is because the output of an operator, which is a set of tuples, can be the input of the next operator, and the performance of an operator depends on the sizes of its inputs. Another, weaker goal of query optimization is preventing the selection of bad query execution plans.

1.1 IMPACT OF QUERY EXECUTION PLAN CHOICE

To demonstrate the impact of the query execution plan choice on the query execution time, Figure 1.1 shows the results of an experiment by Neumann [75]. In the experiment, randomly chosen query execution plans for the Query 5 of the TPC-H benchmark [89] are executed. Although all plans create the same result, Figure 1.1 shows that some query execution plans finish in a few milliseconds while other plans reach the execution time limit of three seconds. This illustrates that the choice of the query execution plan has a strong impact on execution time. In practice, query optimizers cannot guarantee to find fast query execution plans and avoid bad query execution plans [94].

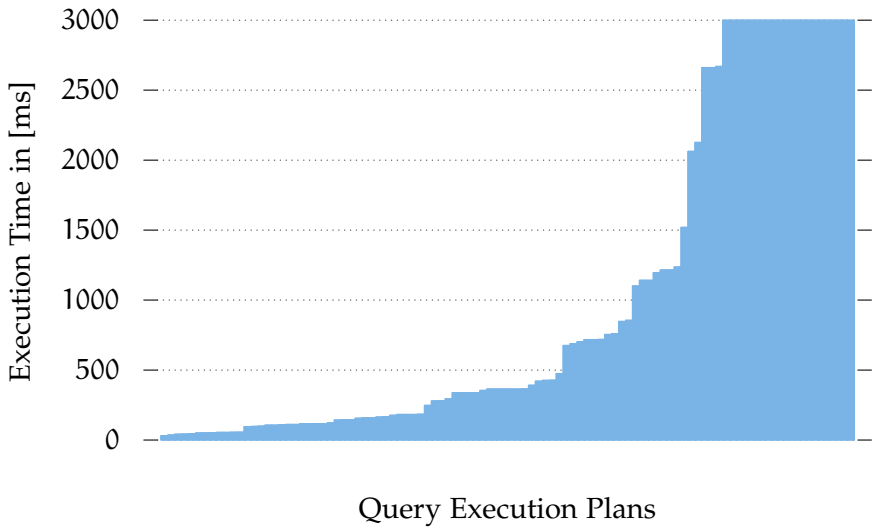


Figure 1.1: Experiment by Neumann [75] showing the query execution times for randomly selected query execution plans for TPC-H Query 5.

1.2 REASONS FOR SUB-OPTIMAL PLAN CHOICE

The seminal work on query optimization by Selinger [4] described a cost-based query optimizer. In essence, cost-based query optimization works as follows: (1) Different query execution plans are enumerated based on some strategy. (2) Costs are assigned to each enumerated plan using statistics, estimations, and some cost functions. (3) The query execution plan with the smallest estimated cost is selected as optimal plan. According to Leis et al. [78, 93], the major reason why query optimizers choose sub-optimal or even bad query execution plans are cardinality estimation errors. Since the estimated cardinalities, i.e., the number of tuples, of intermediate results are central parameters in each cost model, incorrect cardinality estimates create wrong cost estimates and thus can lead to a wrong query execution plan choice. While the performance impact of the cost function is quantified by Lohman to at most 30% [73], cardinality estimation errors are theoretically unbounded. Leis et al. [78, 93] have shown cardinality estimation errors of up to six orders magnitude.

The root causes for cardinality estimation errors can be categorized by assumptions on: (1) the distribution of values within a column, (2) the correlation between columns, (3) the join relation, and (4) the freshness of statistics, which do not always hold. Cardinality estimators conventionally assume a uniform distribution of values within a column, although the data is frequently skewed in practice. Furthermore, cardinality estimators assume *Attribute Value Independence (AVI)* for the correlation between columns within a table, which does not hold in general [73]. The principle of inclusion [93] is another as-

sumptions, which only holds for foreign key joins. Finally, cardinality estimators always assume to work on up-to-date statistics, although statistics can be stale. In 1991, Ioannidis et al. [16] showed that the cardinality estimation error can grow exponentially with the number of performed joins. Despite the effort put into improving cardinality estimation since then, cardinality estimation is still the major weak spot in query optimization, as Leis et al. [78, 93] have recently shown.

In addition to the existing issues of cardinality estimation in join queries, new challenges are introduced by a new kind of workload: *Hybrid Transactional/Analytical Processing (HTAP)* [62, 76, 95] is composed of two fundamentally different workloads: First, transactional workload having many concurrent transactions that read, insert, update and delete a small number of tuples in a table. Second, analytical workload with complex and calculation-intensive queries that scan, filter, join, and aggregate entire tables. While traditional database setups are split into a transactional and an analytical database system, HTAP database systems avoid the delay for loading data from one system into the other, to enable data analytics on fresh data. For the HTAP database systems that means to process the expensive analytical queries on the mission-critical transactional database. This introduces a trade-off between the quality and freshness of statistics, and the effort to calculate and maintain them.

Analytical database systems maintain comprehensive statistics on columns and tables, such as histograms and top-k values, to do best effort cardinality estimations in complex queries. Since analytical database systems face a low number of updates, the effort to create and maintain comprehensive and precise statistics amortizes. In contrast, transactional database systems process less complex queries, face a high number of updates, and focus on high throughput and fast response times. Consequently, transactional database systems tend to have less comprehensive statistics. For HTAP database systems, there are the following three options for statistics: First, comprehensive statistics that are updated with each transaction. This is good for complex analytical queries, but can hurt the transactional throughput and response time guarantees. Second, comprehensive statistics that are created at query optimization time. This is good for the quality of query execution plans and the transactional performance, but can introduce optimization time overheads in complex queries. Third, simple and cheap statistics that can be maintained without hurting the transactional performance. Although this may cause stronger cardinality estimation errors and worse execution plans for complex queries, simple and cheap statistic are the consequent choice to guarantee the transactional performance in HTAP.

To sum it up, it has been shown [78, 93] that complex queries are still a major challenge for conventional database systems, since they can cause cardinality estimation errors during the query optimiza-

tion. In addition, more sophisticated statistics, which may improve the cardinality estimates in complex queries, are unsuitable for HTAP systems with strong requirements for transactional throughput and response time.

1.3 PROBLEM STATEMENT

According to Leis et al. [78, 93], cardinality estimation errors are evident. Improving the cardinality estimation in join queries with a reasonable computation effort to a reasonable level of precision is hard to achieve and requires efforts that are unsuitable for HTAP workloads. We identified the following problems related to cardinality estimation errors, which are only partially solved or yet unsolved. We argue that the following problems should rather be addressed, than trying to further improve the conventional estimation of cardinalities.

1.3.1 *Detection of Sub-Optimal Plans*

The first problem is the impact of cardinality estimation errors on the optimality of query execution plans. It is trivial to detect a cardinality estimation error by comparing the estimated cardinality of an intermediate result and its true cardinality at query execution time. In contrast, it is not trivial to decide if the cardinality estimation error makes the query execution plan sub-optimal or if the plan remains optimal. The trivial solution is the invocation of the optimizer to figure out if a cheaper plan exists, but the corresponding effort can grow exponentially with the number of joined tables. Therefore the question emerges: What is a more effective solution to decide if an optimal plan remains optimal, and what is the complexity of such a solution? We introduce our contribution in Section 1.4.1.

1.3.2 *Awareness of Estimation Errors*

Though cardinality estimation errors are evident [78, 93], they are not considered in the conventional plan selection. Conventional query optimizers still assume that the cardinality estimates are correct and choose the estimated cheapest plan. We argue that estimation errors should be considered during query optimization and introduce our corresponding contribution in Section 1.4.2.

1.3.3 *Integration of Runtime Knowledge*

Superior to cardinality estimates are the true cardinalities of intermediate results, which can be taken at query execution time. The true cardinalities taken at query execution time can be utilized to compen-

sate cardinality estimation errors and consequently improve query execution plan quality. Although runtime feedback is superior to estimations, conventional query optimizers solely rely on estimations, and disregard the potential quality improvements for query execution plans that can be achieved by runtime feedback. We introduce our contribution to improve plan quality by considering runtime feedback in Section 1.4.3. It is an improvement on existing work.

1.4 CONTRIBUTIONS

This work makes contributions in Robust and Adaptive Query Processing. We address unsolved problems and limitations of existing work to detect and compensate sub-optimal query execution plans, which were caused by cardinality estimation errors. According to Lohman [94], "*robust and adaptable query plans are superior to optimal ones*". Relating to that quote, our major contributions can be summarized as:

- An efficient algorithm to calculate the intermediate result cardinalities, denoted as *optimality ranges*, for which a query execution plan is optimal,
- Three metrics to quantify the robustness of a query execution plan towards cardinality estimation errors together with a corresponding robust plan selection strategy, and
- A revisit of *Mid-Query Re-Optimization* [31, 39, 41] including a novel adaptive query processor that enables efficient query re-optimizations and efficient query execution plan switches at query execution time.

Before we explain our contributions in detail, we show the basic architecture of query processors in Figure 1.2, to better see which components of a query processor can be improved by the contributions of this work. Each query processor consists of two major components, which are highlighted in Figure 1.2:

1. The query optimizer, which gets a query and searches a query execution plan corresponding to the optimization goal, e.g., minimal execution time or minimal resource consumption.
2. The execution engine with the implementations of all relational operators such as scans, filters, joins, and aggregations, and a mechanism to pass tuples from one operator to another operator. Each query execution plan is passed from the query optimizer to the query execution engine. The query execution engine calculates the query result and returns it.

We further discuss the foundations of query optimizers and query execution engines in the Sections 2.5 and 2.6.

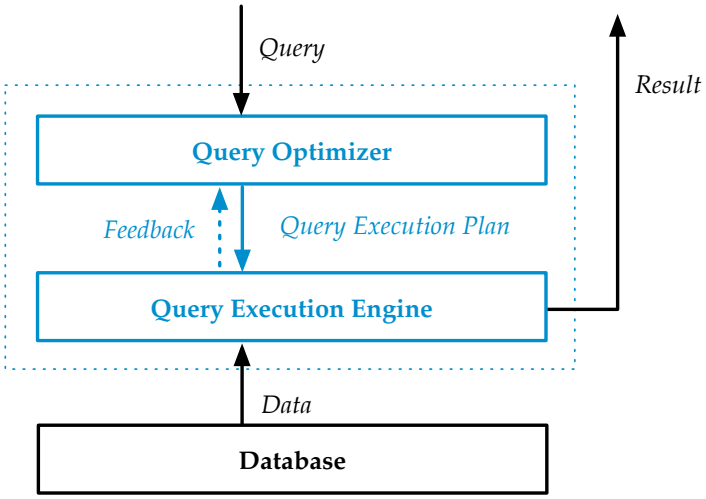


Figure 1.2: Basic architecture of query processors.

1.4.1 *Precise Optimality Ranges*

Optimality ranges for the cardinality of intermediate results enable to efficiently detect if a query execution plan became sub-optimal due to a cardinality estimation error. Optimality ranges reveal for which cardinalities an 'optimal plan' remains optimal. An optimality range has a lower and an upper bound for the cardinality of an intermediate result. Within this cardinality range, the plan is optimal. Optimality ranges can be calculated for each intermediate result in the query execution plan. To the best of our knowledge, the calculation of precise optimality ranges is an unsolved problem. Existing approaches cannot characterize optimality ranges precisely, because they rely on simple heuristics [31, 41], or consider only a subset of the required plan alternatives [39].

OUR CONTRIBUTION: In Chapter 3, we present an algorithm to calculate precise optimality ranges, which takes all necessary plan alternatives into account. We also describe an effective pruning strategy that keeps the number of enumerated plans during optimality range calculation low. Further, we share a formal and experimental analysis on the complexity of optimality range calculation. In addition, we present an experimental analysis on optimality ranges for a TPC-H-like benchmark, showing the missing precision and limitations of heuristic approaches to characterize optimality ranges. Optimality ranges can be calculated on optimal query execution plans between query optimization and execution (see Figure 1.2). The algorithm can be implemented as extension of the optimizer, and utilize data structures and work that has been done during the initial optimization. It supports all kinds of queries, operators and tree structures of query execution plans. We explain the applications of opti-

mality ranges, such as execution plan caching and parametric queries in Section 3.5. The calculation of optimality ranges is the foundation for our contributions, which we introduce in Section 1.4.2 and 1.4.3. We previously published parts and ideas of our work on optimality ranges in Wolf et al. [100].

1.4.2 *Robustness Metrics and Robust Plan Selection*

To consider potential cardinality estimation errors during the query optimization, we rely on *Robust Query Processing* [80], or more specifically *Robust Plan Selection*. The goal of Robust Plan Selection is to find and execute query execution plans, which are less sensitive to estimation errors of cost parameters, such as cardinality. A robust plan does not significantly change its execution time or resource consumption in case of estimation errors. Existing approaches for robustness plan analysis and selection require considerable computation effort and cannot be applied at optimization time [53, 71, 82]. The optimization time approaches are limited to certain tree structures of query execution plans [77, 81], support only plans that are optimal for some cardinalities [40, 41, 53, 82], or need to see alternative plans to quantify the robustness of the considered plan [59, 77, 81].

OUR CONTRIBUTION: In Chapter 4, we present a robust plan selection strategy based on new robustness metrics, which can be calculated efficiently at optimization time. We give a formal problem description and consistency requirements for robustness metrics. Based on that, we present three novel metrics to quantify the robustness of query execution plans towards cardinality estimation errors. While cost models assign estimated cost to each given plan, our robustness metrics assign an estimated robustness value to each given plan. The metrics do not need to see alternative plans to quantify the robustness of the considered plan. Since the calculation of our robustness values is cheap and can be done at optimization time, we use them for our plan selection strategy. Our robust plan selection takes both, estimated cost and estimated robustness into account, to find a plan that is cheap and robust. The robustness metrics and robust plan selection can be applied to any kind of query containing any kind of operator, and supports all kinds of tree structures of query execution plans. Further, we do not require the query execution plans to be optimal somewhere. Integrating our contributions into a query processor requires minimal modifications, because it affects only the query optimizer. The robust plan selection is still based on estimations, and not on runtime feedback. Nevertheless, our experimental evaluation shows that our robust plan selection achieves faster query execution times in presence of cardinality estimation errors, compared to the conventional plan selection that picks the estimated cheapest plan.

We previously published parts and ideas of our work on robustness metrics and robust plan selection in Wolf et al. [101]. The master thesis of Brendle [97] is based on the initial submission of Wolf et al. [101]. While Brendle contributed the experiments, my contributions are the consistency requirements for robustness metrics, the robustness metrics themselves, and the robust plan selection strategy. We ran the experiments again with some considerable improvements for the camera ready version of Wolf et al. [101] and this work.

1.4.3 *Mid-Query Re-Optimization Revisited*

Superior to estimations are the true statistics, taken from the calculated intermediate results at execution time. Unfortunately, they are only available after the calculation of intermediate results. *Adaptive Query Processing* [47, 80] is a subset of Robust Query Processing and utilizes these true statistics to improve the query execution plan quality. One Adaptive Query Processing approach is called Mid-Query Re-Optimization. It invokes the query optimizer at execution time with runtime statistics to compensate potential estimation errors, and switches to a better query execution plan. Existing Mid-Query Re-Optimization approaches [31, 39, 41] try to keep query processor modifications minimal, and focus on criteria for triggering re-optimizations, because they can cause a considerable overhead.

OUR CONTRIBUTION: In Chapter 5, we present a novel Mid-Query Re-Optimizer. In order to speedup re-optimizations, it contains a selective re-enumeration algorithm that enables efficient query re-optimizations at execution time. Furthermore, we describe an improved Mid-Query Re-Optimization strategy, which takes the insights from the experimental evaluation on optimality range calculation into account. We include these building blocks into an architecture of an adaptive query processor. The key design aspect of the adaptive query processor is a tight integration of optimizer and execution engine that enables efficient plan switches at execution time. Consequently, it requires significant changes to the query optimizer and the execution engine, including a feedback loop from the execution engine to the optimizer (see Figure 1.2). Our adaptive query execution strategy supports all kinds of queries, operators and tree structures of query execution plans. It furthermore improves already the first execution of a query, which includes ad-hoc queries, because it does not need multiple query executions to learn from [36]. We demonstrate in the experimental evaluation that our query processor with adaptive execution can outperform the conventional execution of a static plan.

2 FOUNDATIONS

This chapter gives an overview of the foundations of robust and adaptive query processing. We start with clarifying fundamental notations in Section 2.1. We continue with explaining some essential concepts of database systems, namely the different workloads in Section 2.2, the impact of the memory hierarchy in Section 2.3, and the different database storage layouts in Section 2.4. As we discussed in Section 1.4, each query processor consists of two major components: Section 2.5 explains the query execution engine, which contains the implementations of all relational operators such as scans, filters, joins, and aggregations, and a mechanism to pass tuples from one operator to another operator. Section 2.6 explains query optimizer, which searches query execution plans corresponding to the optimization goal. Our focus in this work is on minimal query execution time. We summarize the foundations in a query processor architecture presented in Section 2.7, which we use to implement our approaches and do experimental evaluations.

2.1 NOTATIONS

Throughout this work, relations or tables are denoted by a single capital letter starting from R. Arbitrary relations or tables in proofs are denoted by R and an additional subscript, such as R_i or R_j. Relations and tables of benchmarks keep their specified name. The query execution plans are denoted by P, having a subscript to further describe the plan, such as the optimal plan P_{opt}. Each edge in a query execution plan connects two operators, and therefore represents an intermediate result. For variable quantities like the cardinalities of intermediate results f, the selectivities of operators s, and the costs of plans or sub-plans c, we denote estimations as $\hat{f}$, $\hat{s}$ and $\hat{c}$, and true values as $\mathring{f}$, $\mathring{s}$ and $\mathring{c}$. Cardinalities have an additional subscript describing the edge or intermediate result. For example, the estimated cardinality on the edge, which represents the result of joining R and S is denoted by $\hat{f}_{R \bowtie S}$ or $\hat{f}_{RS}$. In proofs, the cardinality of an arbitrary edge e is denoted by f_e. Selectivities have a subscript that describes the associated operator. Cost functions to calculate costs are denoted by C, having a subscript to describe the cost function.

2.2 WORKLOADS

The traditional workload in database systems is *Online Transaction Processing* (OLTP). It consists of a large number of mostly short running transactions, which are sequences of read, insert, update, and delete operations. Transactions touch only a small number of tuples in the database. In conventional relational database systems, transactions have the *ACID*-properties [6]: *Atomicity*, *Consistency*, *Isolation*, and *Durability*. A transaction can consist of multiple statements. Atomicity guarantees that each transaction is an individable unit, which is either completely committed or completely aborted. Consistency ensures that each transaction transfers the database from a consistent state into another consistent state. Isolation guarantees that concurrent transactions are isolated from each other, and create the same database state as a sequence of these transactions. Durability ensures that successfully committed changes of a transactions are persistent.

With the increased popularity of analyzing data, *Online Analytical Processing* (OLAP) extended the workload in database systems. On the one hand, OLAP workloads are read-mostly, which requires less synchronization compared to OLTP workloads. On the other hand, a single OLAP query takes considerably more computation time compared to a single OLTP transaction, because it can scan, filter, join, and aggregate entire tables. OLAP systems are also called *Data Warehouses* or *Decision Support Systems*. In conventional database setups, data is regularly extracted, transformed, and loaded from the mission-critical OLTP database into the OLAP database to be analyzed.

The demand of business applications to get decision support in real-time formed a composite workload of OLTP and OLAP, called *Hybrid Transactional/ Analytical Processing* (HTAP). Compared to a setup consisting of OLTP and OLAP system, HTAP systems avoid the delay for loading data from one system into the other. This enables analyzing always the freshest data, and immediately detect issues in a business process. As we have shown [76], HTAP workloads require a sensitive scheduling between the large number of short-running OLTP transactions, and the fewer but more expensive OLAP queries.

2.3 MEMORY HIERARCHY

The hierarchy of memory has a strong impact on the architecture of database systems, and advances in memory technology periodically open new research questions and opportunities to further increase performance and capacity of database systems. Hennessy and Patterson [92] give a comprehensive overview of memory technologies and the memory hierarchy. Memory is smaller, faster, and more expensive the closer it is to the processors. The smallest, fastest and most expensive pieces of memory are the processor caches, which share a

wafer with the processor cores. The next major layer in the memory hierarchy is the main-memory mostly based on DRAM technology, followed by persistent hard disk drives or solid-state drives. Conventional database systems store the database and intermediate results of queries on disk, and utilize the main-memory as cache. Through the advances in DRAM technology, modern database systems can process queries mostly in-memory to avoid the storage of intermediate results on disk. In addition, systems such as SAP HANA [95] can load their working set of tables into main-memory to avoid reading from disk. They use the disk for the primary database and the persistent transaction log to achieve durability. A related research question for in-memory systems is the strategy to offload data that is not needed, i.e., cold data, from main memory to disk [86]. Since main-memory is more expensive than disks, another research question considers the size of the main-memory that is economically reasonable: According to the *Five Minute Rule* [11], the size of the main-memory should enable to keep the working set of data for five minutes in main-memory. The Five Minute Rule was revisited every ten years to consider technological and economical advances [28, 48, 87].

To scale-up the main memory size and the number of processor cores, modern scale-up systems are divided into *Non-Uniform Memory Access* (NUMA) nodes [92]. Each NUMA node has some cores and one memory controller. NUMA architectures can limit the scalability of data-intensive applications such as database system: Since there is one global memory space, a process running on one NUMA node can access the memory on another NUMA node. As a result, it faces a different memory access latency compared to a memory access on the local memory controller. Ignoring NUMA architecture in the implementation of database systems can limit the scalability.

2.4 STORAGE LAYOUTS

Traditional database systems store data and index structures in pages, whose size corresponds to the block size on disk. Some main-memory systems, such as SAP HANA, do not completely rely on pages and can directly allocate main-memory corresponding to the size of their data structures. There are two major interpretations for the physical storage of the relational model [1], i.e., the storage of database tables in pages or main-memory: First, the row format, which stores a table tuple by tuple. Second, the column format, i.e., the *Decomposed Storage Model* [7], which stores a table column by column, and consequently single attributes of a tuple not physically together.

For OLTP workloads, i.e., reading, inserting, updating, and deleting only a few rows in a table, the row layout is superior, because only a consecutive piece of memory and not different pages or non-consecutive memory pieces have to be accessed to process a single

row. For OLAP workloads, i.e., scaning, joining and aggregating entire tables, the row layout turned out to be unsuitable: OLAP queries usually read only a subset of the columns in a table. Consequently, the row layout results in reading columns from disk or main-memory that are not needed. In the column layout, only the necessary columns have to be read. Storing data in a column layout has further advantages, because it enables more efficient table compression algorithms.

There are also hybrid layouts such as PAX by Ailamaki [35] or HYRISE by Grund [61], which are more suitable for HTAP workloads. PAX for instance stores multiple complete tuples together in a page such as the row format does. Inside a page, the tuples are stored in the column format. In this work, we use the column format, because we focus on the processing of complex join queries. Furthermore, the analytical workload dominates the resource consumption in HTAP systems [76], and should be processed as efficient as possible.

2.5 QUERY EXECUTION ENGINES

Driven by the latest achievements in processor and memory architectures, the design and implementation of query execution engines is an ongoing challenge. Query execution engines contain the implementation of operators, and the passing of data between operators in the query execution plan.

2.5.1 *Join Operator*

Most research publications focus on *equality-joins* or *equi-joins* [63, 66, 70, 85, 88], which join tuples that have equal values on certain columns. There are also other kinds of join predicates such as *in-equality*. Join operators without a join predicate are denoted as *cross-products* or *cross-joins*, because each tuple of one relation is joined with all tuples of the other relation. Most joins-operators join two relations, but there are also multi-way joins, joining more than two relations, which can be split into multiple binary joins. Joining tables can consume the majority of the query execution time. Consequently, there is a large number of publications, which propose join algorithms that are optimized for certain join predicates, number of joined relations, data characteristics or special system architectures [85, 88].

In general, a query processor can have different implementations for an operator, and the query optimizer has to choose the best. For binary joins, there are three major categories of implementations [18]: First, *nested-loop joins*, which iterate for each tuple in the outer relation over all tuples in the inner relation, to identify tuples that can be joined. An improvement of nested-loop joins are *index-nested-loop joins* that use an index on the inner relation, to avoid a full scan of the inner relation for each tuple in the outer relation. Second, *hash joins*

that build a hash-map on the smaller, inner relation, and do a hash table lookup for each tuple in the outer relation. Hash joins support only equality join predicates. Throughout this work, the build side of hash joins is depicted on the right-hand side. Third, *sort-merge joins* which take relations sorted on the join columns as input, and perform a merge to figure out which tuples can be joined. Some sort-merge join implementations accept unsorted inputs and and have efficient implementations to sort the unsorted inputs [88].

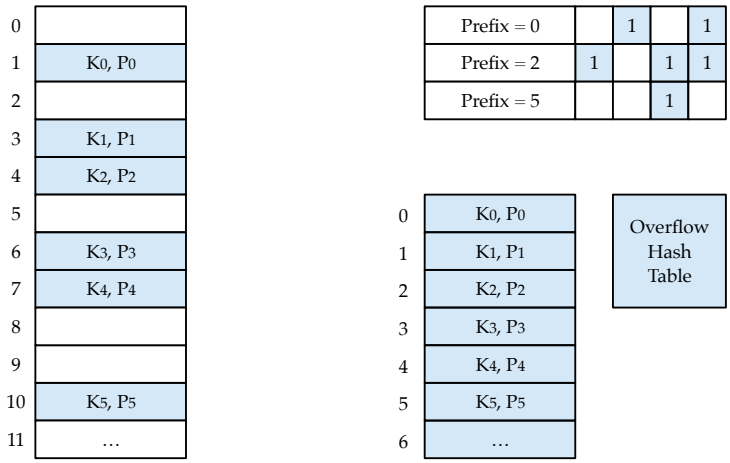
For *inner joins*, a match between two tuples results in the creation of an output tuple. Tuples without a join partner are not in the join result. Next to inner joins there are also *semi-*, *anti-*, and *outer joins*.

It is an ongoing discussion which equi-join implementation is the fastest [63, 66, 85]. In general sort-merge joins are better suited for joining relations with similar cardinality, and hash-join for joining relations with a considerable cardinality difference. Since the cardinality estimation errors and other topics in this work are independent of the join implementation, we use only one join operator implementation in this work. Since memory consumption is crucial for in-memory systems, we took the state-of-the-art memory efficient hash join, the *Concise Hash Table* (CHT) join [70], and further improved it.

2.5.1.1 *Concise Hash Table Join*

We extended the original CHT join [70] for multi-socket systems, and query execution plan switches between build and probe phase. We explain the details of these extensions together with our adaptive join processor in Chapter 5. The basic CHT [70] was designed on the following assumptions: The size of hash tables is over-provisioned to decrease the number of hash collisions. Consequently, multiple buckets in the hash table stay empty and memory is wasted as depicted in Figure 2.1a. A CHT in contrast splits the hash table into a bitmap to indicate if a bucket is filled, and a dense payload table that holds the key and the payload. Figure 2.1b shows the bitmap on the upper side and the payload table on the lower side. Since the bitmap just consumes one bit per bucket, it can be over-provisioned without wasting too much memory. The CHT is a linear probing hash table that probes the actual bucket and the following bucket. Entries for which no free bucket is found are stored in an overflow hash table. Compared to a conventional hash table, filling a CHT requires two input scans.

The first scan fills the bitmap, which indicates if a bucket is filled and enables an efficient calculation of the index in the payload table. The index in the payload table is the number of filled buckets up to the current bucket. To efficiently calculate this number without scanning the entire bitmap, the CHT uses a prefix bitmap. The prefix bitmap is an array of 64-bit words, as shown in the top of Figure 2.1b. In each of 64-bit word, 32 bits are used for the actual bitmap, and 32 bits to store the number filled buckets until the 64-bit word, i.e., the



(a) Logical (b) Physical

Figure 2.1: Concise Hash Table [70].

prefix. Consequently, the index in the payload table is the index in the current word plus its prefix. After filling the bitmap in the first scan, the prefixes are calculated. Next, the data is scanned again, the prefix bitmap is used to calculate the index, and key and payload are inserted at the corresponding index in the payload table.

Probing into the CHT starts with a lookup into the prefix bitmap. If the bucket is filled, the index in the payload table is calculated and the key of the corresponding entry in the payload table is checked for equality. If all linear probing buckets are filled but no key matches, the overflow hash table has to be probed to check if an entry exists. We discuss the parallelization of operators and the parallelization of the CHT join in particular in Section 2.5.3.2.

2.5.2 *Pipelining*

Another performance-critical aspect next to the operator implementation is passing of intermediate results between operators. There are two dimensions by which the passing of intermediate results can be categorized: *granularity* and *direction*. The granularity describes the number of tuples that are passed from one operator to another. It distinguishes between passing a *tuple-at-a-time* [22], a batch of tuples at a time, i.e., *batch-at-a-time* [42], or the entire intermediate result. Tuple-at-a-time and batch-at-a-time are denoted as *pipelining*. The direction, distinguishes between the *pull* [22] and the *push* model [34]. It describes if tuples are recursively pulled from the top of the plan, or pushed from the bottom of the plan. Next, we further explain these dimensions by discussing the seminal pipeline designs, and categorize the designs by granularity and direction. The most trivial approach is passing the entire intermediate result from one operator to another. It

means to materialize the entire intermediate result after every operator to disk or main-memory. Consequently, every operator has to read from disk or main-memory and cannot utilize data cache locality.

Another option to pass data from one operator to another is pipelining [3]. Graefe [22] described a pipeline of operators, in which single tuples are passed from one operator to another, i.e. tuple-at-a-time granularity. Graefe implemented a recursive iterator pattern, where tuples are requested from the final operator in the query execution plan, i.e., the pull model. In the pull-model, a tuple request on an operator causes a recursive request on its child or children. Pipelining is great for data cache locality and works for operators that are no pipeline breakers such as filter operators. A filter operator for instance requests single tuples from its child, until a tuple satisfies the filter condition and can be returned. Other operators such as sorting or aggregation are pipeline breakers, because they need the entire input before returning an output tuple. A hash join has two children, build side and probe side. The build side of the hash join, is a pipeline breaker. Consequently, the first tuple request on such an operator fetches all tuples from the build side to build the hash table. Next, the join operator requests tuples from the probe side until a tuple matches an entry in the hash table and can be returned, or the probe side is completely processed. The probe procedure is repeated for all following tuple requests on the hash join operator.

Boncz [42] observed that passing single tuples through a pipeline causes a virtual function call every time a tuple is passed from one operator to another, and consequently a substantial overhead and a bad instruction cache locality. To address these shortcomings, Boncz proposed to pass a batch of tuples at a time, i.e., batch-at-a-time or *vector-at-a-time*, instead of a single tuple at a time. Essentially this is a trade-off between no pipelining, i.e., materializing entire intermediate results after each operator, which results in losing the data cache locality, and passing a tuple at a time, which results in a virtual function call overhead. Further, passing a batch at a time enables *data-level parallelism* such as *Single Instruction, Multiple Data (SIMD) vectorization* [92], to simultaneously process tuples within an operator. The pipeline described by Boncz [42] implements the pull model.

Kemper and Neumann [62] showcased a more recent pipelining approach that uses the push-model. In the push-model, tuples are pushed from the bottom of the plan through the pipelines to a pipeline breaker, e.g., a sort operator or the build side of a hash join. For query execution plans with multiple pipeline breakers, the pull-model implicitly handles the dependencies of pipelines. In the push-model, the execution order of pipelines has to be orchestrated explicitly, which creates opportunities for adaptive query processing [69]. Apart from that, the push-model enables a more simplistic implementation of parallelism within a single pipeline. The base table at the beginning of

a pipeline can be split into partitions of arbitrary size, and they can be pushed through a pipeline in parallel without any synchronization. We further discuss intra-pipeline parallelism in Section 2.5.3.1. The pipelining by Kemper and Neumann [62] utilizes just-in-time compilation. For each pipeline and all its operators it creates a piece of code that is compiled and executed at query execution time. On the one hand, just-in-time compilation completely avoids virtual function calls, further improves instruction cache locality, and decreases branching. On the other hand, it has a high implementation complexity and introduces an additional compilation overhead. The initial approach by Kemper and Neumann uses the tuple-at-a-time approach since virtual functions calls do not occur anymore. To address the scalability issues on NUMA systems [72], and speedup the parallel execution of pipelines with SIMD vectorization [98], they use the batch-at-a-time approach. Just-in-time compilation of pipelines results in a push-model, because each pipeline is a function that takes the input relation and materializes the result in the pipeline breaker.

We use the batch-at-a-time pipelining in this work, because it is a reasonable trade-off between performance and implementation complexity. Further, we use the push-model, due to its advantages for intra-pipeline parallelization that we further discuss in Section 2.5.3.1.

2.5.3 *Parallelization*

The parallelized execution of queries in database systems has a major performance impact. Hennessy and Patterson [92] give a comprehensive overview of parallelization techniques in computer architectures. In modern scale-up systems, parallelization is achieved by *task-level parallelism* such as parallel threads, by *data-level parallelism* such as *Single Instruction, Multiple Data (SIMD) vectorization*, and of course implicitly by *instruction-level parallelism*.

In database systems, different queries can be processed simultaneously, i.e., *inter-query parallelism*, which increases the query throughput. In addition to that, parallelism can be also achieved within a single query, i.e., *intra-query parallelism*, which improves the execution times of single queries. One level of intra-query parallelism is the simultaneous execution of independent pipelines of a query execution plan. It can be denoted as *inter-pipeline parallelism*. Inter-pipeline parallelism cannot utilize modern scale-up systems with multiple processor sockets and hundreds of cores, when only a small number of queries is processed simultaneously in the system.

2.5.3.1 *Intra-Pipeline Parallelism*

To increase the level of parallelism in modern scale-up systems, tuples have to be pushed in parallel through a single pipeline. In theory, every single tuple could be pushed in parallel through a pipeline,

e.g., by a thread, but in practice this creates a parallelization overhead. Therefore, it is more reasonable to push batches of tuples [72].

To address the scalability issues in NUMA-systems, Leis, Boncz, Kemper, and Neumann [72] created a NUMA-aware strategy to parallelize single pipelines. It splits tables into batches of tuples, denoted as *morsels*. The morsels are allocated on a NUMA node, and pushed by a thread that is bound to the NUMA node through the pipeline. This concept is an extension of the batch-at-a-time approach [42], and ensures that remote data accesses are reduced to a minimum. We use the NUMA-allocated morsels and the NUMA-aware workload scheduling [72] for intra-pipeline parallelism in this work.

2.5.3.2 *Intra-Operator Parallelism*

Orthogonally to pushing tuples in parallel through a pipeline, there are also strategies to parallelize pipeline-breaking operators such as join, sort and aggregation. Schuh [85] gives a comprehensive overview of parallel hash and sort-merge join algorithms including the CHT join. One option for building a hash table in parallel is partitioning the input by hash value, and build the partitions in parallel. Other options are non-partitioning hash joins, which use synchronization to build a single global hash table. The probe side of a hash join is no pipeline breaker, and probing into a hash table can be done in parallel without synchronization. There are hash join algorithms that also partition the probe side to achieve cache efficiency [66]. In sort-merge joins, all unsorted inputs are pipeline breakers, because they have to be sorted before the merge. Sorting a single input can be done in parallel by partitioning the input, and sorting the partitions in parallel.

Aggregation operators can be also implemented using hashing or sorting. Müller [79] presented a cache efficient aggregation operator, which operates at the optimal number of loaded cache-lines of sort- and hash-based aggregation. It has been designed to run on modern scale-up systems, and uses a recursive hash-based partitioning to enable the full parallelization of all phases of the aggregation algorithm.

2.6 QUERY OPTIMIZERS

The objective of query optimizers is to select a good query execution plan from a set of plans that can grow exponentially with the number of joined relations. We focus on cost-based query optimization, which enumerates different plans based on some strategy, calculates the cost for each plan using statistics, estimations and a cost function, and finally selects the plan with the cheapest estimated cost. Furthermore, we consider enumeration algorithms that exhaustively enumerate the plan space, and have no limitations to certain execution plan structures such as left-deep or linear trees.

2.6.1 *Enumeration Algorithms*

The initial publication on query optimization was a seminal work by Selinger et al. [4] describing a dynamic programming based enumeration algorithm called *DPsize*. Dynamic programming is based on Bellman's principle of optimality [57] which says that an optimal solution can always be constructed from optimal sub-solutions. In practice, enumeration algorithms keep all plans that are not dominated by another plan. A plan dominates another plan, when it is cheaper and has at least the same properties. If there are two plans, and one plan is cheaper, but the other plan has more properties, the enumeration algorithm keeps both plans. There are logical properties of a plan, such as the referenced tables. Plans with the same logical properties form a plan class. Plans can also have physical properties such as sorted columns. Physical properties can be utilized by the following operators. As a consequence enumeration algorithms do not only keep the cheapest plan per plan class, but additional more expensive plans that have different physical properties. If an optimal plan is constructed from sub-optimal plans with physical properties, Bellman's principle of optimality does not hold anymore.

Each dynamic programming enumerator has a dynamic programming table, which has an entry for each plan class that contains at least an optimal plan for this plan class. DPsize enumerates plans by size, i.e., the number of joined tables, starting with plans of size one, and continuing to the final solution size. To enumerate plans of a certain size, it combines the optimal plans of sub-plan classes. If cross joins between two relations are not allowed, it checks if the plans are feasible, i.e., if there is at least one join attribute that connects both sub-plans. Finally, the optimal plan in the plan class that represents the entire query is selected as optimal plan. Whether the identified plan is the true optimal plan in practice depends on the quality of the cost model, and the quality of the cardinality estimations as a core component in the cost model.

To better analyze the behavior of query optimization algorithms, queries are categorized by their topology: Figure 2.2 shows some typical topologies that are considered. Chain and Cycle queries are the simplest topology, and usually require the least optimization effort. The typical data warehouse queries have a star or snowflake topology. They have a significantly larger number of plan alternatives compared to chain or cycle queries that join the same number of relations, and consequently need a higher optimization effort. Clique queries do almost not occur in practice, but are theoretically the worst case in terms of plan alternatives and optimization effort. When query execution plans with cross joins are enumerated, their enumeration complexity is equal to a clique queries that reference the same number

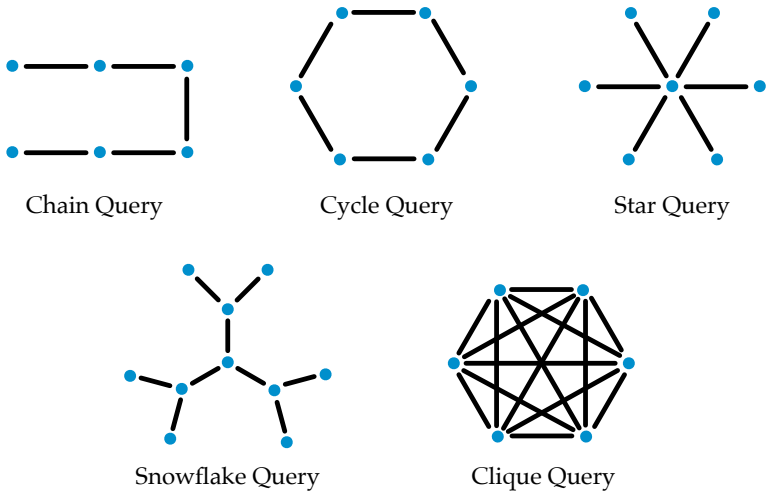


Figure 2.2: Overview of typical join query topologies.

of tables. In practice, queries can be a combination of the theoretical topologies, containing chains, multiple cycles, and small stars.

Ono and Lohman [14] analyzed the complexity of DPsize join enumeration for queries with chain and star topologies. They showed that the number of feasible plans, i.e., without cross products between relations, grows polynomially for chain queries and exponentially for star queries with respect to the number of joined relations.

Vance and Maier presented a dynamic programming algorithm called *DPsub* [27], which is superior to DPsize for star and clique topologies. It essentially iterates over all plan classes and very efficiently enumerates all sub-plan combinations for each plan class.

Moerkotte and Neumann experimentally evaluated the complexity of DPsize and DPsub, and proposed a new dynamic programming algorithm that only enumerates the feasible plans or connected subgraph complement pairs, and called it *DPccp* [44]. DPccp is superior to DPsize and DPsub for chain, cycle, star and clique queries, because DPccp enumerates only feasible plans. DPsize and DPsub enumerate a larger number of plans and check if they are feasible.

While DPsize, DPsub and DPccp enumerate plans bottom-up, DeHaan et al. [46] proposed a top-down dynamic programming enumerator. Top-down enumeration uses memoization to save the optimal plan for a visited plan class and avoid redundant searches. Compared to bottom-up enumeration, top-down enumeration identifies a complete plan and its cost much earlier. The cost of a complete plan can be used to prune more expensive sub-plans during enumeration, since the resulting plan cannot be cheaper. Top-down enumeration enables further pruning. The cost of an optimized sub-plan can be used to derive a cost limit for the complementary sub-plan.

For bottom-up enumeration, Moerkotte and Neumann published an improved version of DPccp, which supports more complex join

predicates that involve more than two relations, i.e., queries that are hyper-graphs, and called it *DPhyp* [54]. Finally, Fender and Moerkotte [67] extended top-down join enumeration for hyper-graphs.

Next to dynamic programming, there are other deterministic approaches, such as transformation-based [19, 29] or greedy [30, 57] optimization, and probabilistic approaches such as genetic [12, 15] or randomized [13, 21] optimization. There are also hybrid approaches composed of different optimization algorithms to support very large join queries [99]. Dynamic programming optimization performs an exhaustive plan search, i.e., guarantees to find an optimal solution with respect to the given inputs. Transformation-based optimization can enumerate the same plan space as dynamic programming, if it has corresponding transformation rules. The calculation algorithm for optimality ranges in Chapter 3 requires deterministic decisions during optimization, and consequently works with dynamic programming and transformation-based optimization. The approaches in Chapter 4 and 5 work with arbitrary optimization strategies. We use dynamic programming optimization (DPsize) without cross products in this work, since it supports general tree structures.

2.6.2 *Cost Models*

A core component of cost-based query optimization strategies is the cost model. It assigns cost to a given plan, so that a better plan corresponding to the optimization goal, gets smaller cost compared to another plan. To find a good plan for execution, the costs of enumerated plans are calculated based on statistics and estimations, hence they are denoted as *estimated costs*. To analyze the optimizer behavior and the correlation between cost function and execution engine, costs can also be calculated based on true statistics that are collected during the query execution. Accordingly, they are denoted as *true costs*. A major component in each cost function is the cardinality of intermediate results. Estimating the cardinality after a few number of joins to even a moderate degree of accuracy is still an unsolved problem [78].

In this work, we use two cost functions. First, C_{out} [26] which sums up the cardinalities of all intermediate results in the plan, and implements the intuition of minimizing intermediate result cardinalities:

$$C_{out}(P) = \begin{cases} |R| & \text{if } P \text{ is a base table } R \\ |P| + C_{out}(P_1) + C_{out}(P_2) & \text{if } P = P_1 \bowtie P_2 \end{cases}$$

C_{out} has the *ASI-property* [26], but does not consider different operator implementations and operator-specific decisions, such as the build and probe side of hash joins. As second cost function we use an extension of C_{out}, called C_{mm} [93]. It models the operator implementation as well as the build/probe-side decision of hash joins,

and was created for main-memory systems. Such as C_{out}, it sums up the cardinalities of intermediate results, but for each hash join operator, it also adds the cardinality of the build side. There are other cost functions, e.g., for disk-based [10, 78] or distributed [9, 50] database systems, which have different data access characteristics compared to single node or main-memory database systems. Since we focus on single node main-memory systems, we choose C_{out} and C_{mm}. Furthermore, Leis et al. [78, 93] have shown that simple cost functions such as C_{out} and C_{mm} can achieve strong correlations to the execution engine, and that cardinality estimation is the more severe issue in query optimization. We verify that C_{out} and C_{mm} correlate with our execution engine's behavior in Section 2.7.

2.6.3 *Cardinality Estimation*

The estimation of the cardinality, i.e., the number of tuples, of an intermediate result is a central parameter in every cost function. On the one hand, there are techniques that enable considerably precise estimates, such as histograms for the output cardinality of filter operators. On the other hand, there are operators whose output cardinality is hard to predict, especially in complex queries after many operators. For join operators, Ioannidis et al. [16] have shown that the cardinality estimation error can grow exponentially with the number of performed joins. Despite the effort for improving cardinality estimation since then, Leis et al. [78, 93] have shown that cardinality estimation for join operators is still the major problem in query optimization.

The basic textbook procedure to estimate cardinality is profile propagation [74]. In profile propagation, each column of a table or attribute A of a relation R has a profile $b_A = [\hat{l}_A, \hat{u}_A, \hat{d}_A, \hat{f}_R]$, with $\hat{l}_A$, the estimated minimum value of A, $\hat{u}_A$, the estimated maximum value of A, $\hat{d}_A$, the estimated number of distinct values in A, and $\hat{f}_R$, the estimated cardinality of R. Profiles can have more statistics, e.g., the number of null values in a column. Furthermore, there is a set of rules for different operators to modify or propagate a profile when it was affected by an operator. An exact match filter on a unique column such as $A = x$ results in a propagated profile $b'_A = [\hat{l}'_A, \hat{u}'_A, \hat{d}'_A, \hat{f}'_R]$, where $\hat{l}'_A = x$, $\hat{u}'_A = x$, $\hat{d}'_A = 1$, $\hat{f}'_R = 1$, if A contains x. For an equality join on the attributes A in relation R and B in relation S, the precondition is that $\hat{l}'_A = \hat{l}'_B$ and $\hat{u}'_A = \hat{u}'_B$, which can be achieved by applying the corresponding propagation rules for filters [74]. Applying the precondition filters will also create new estimates for cardinalities of both relations, namely $\hat{f}'_R$ and $\hat{f}'_S$, and new estimates for the distinct counts, namely $\hat{d}'_A$ and $\hat{d}'_B$. Next the output cardinality $\hat{f}_{RS}$ can be calculated using the minimum value $\hat{l}'_A = \hat{l}'_B$, the maximum value $\hat{u}'_A = \hat{u}'_B$,

both cardinalities $\hat{f}'_R$ and $\hat{f}'_S$, and both distinct counts $\hat{d}'_A$ and $\hat{d}'_B$. For the final equation, the distinct counts are not needed:

$$n = \hat{u}'_A - \hat{l}'_A + 1 \tag{2.1}$$

$$\hat{f}_{RS} = \sum_{i=1}^{n} \frac{\hat{f}'_R}{\hat{d}'_A} \frac{\hat{f}'_S}{\hat{d}'_B} \frac{\hat{d}'_A}{n} \frac{\hat{d}'_B}{n} = \frac{\hat{f}'_R \hat{f}'_S}{n} \tag{2.2}$$

In addition, the join creates new estimates for the distinct counts $\hat{d}''_A$ and $\hat{d}''_B$ [74]. Consequently, the profiles for the attributes A and B of relation RS are $b'_A = [\hat{l}'_A, \hat{u}'_A, \hat{d}''_A, \hat{f}_{RS}]$, and $b'_B = [\hat{l}'_B, \hat{u}'_B, \hat{d}''_B, \hat{f}_{RS}]$.

In this work, we assume that the operator selectivity is independent of the the position of the operator in the query execution plan. Therefore, we calculate the selectivity of each operator on the base table level using the minimal profiles described above. Consequently, we are not propagating profiles through the query execution plan. We show in Section 2.7 that the resulting cardinality estimations are competitive to the cardinality estimations in open source and commercial database systems. The selectivity of an operator is defined as the quotient of its estimated output cardinality and its maximum possible output cardinality. For join operators, we use the estimated output cardinality from Equation 2.2 to calculate the selectivity:

$$\hat{s}_{AB} = \frac{\hat{f}_{RS}}{\hat{f}_R \hat{f}_S} \tag{2.3}$$

Consequently, when the corresponding operator joins two base tables or intermediate results with estimated cardinality $\hat{f}_R$ and $\hat{f}_S$, its estimated output cardinality is:

$$\hat{f}_{RS} = \hat{s}_{AB}\ \hat{f}_R\ \hat{f}_S \tag{2.4}$$

An example for the estimation of cardinalities for different intermediate results in a query execution plan is contained in Section 2.6.4. In case multiple join conditions can join two base tables or intermediate results, there are different assumptions to calculate the overall join selectivity. One option is assuming independence of join conditions, and consequently multiply the single join selectivities [74]. Another option is that join attributes are not independent [20, 23–25], which we assume in this work. Consequently, the number of tuples that result from the join is larger compared to independent join attributes. In this case, the correlation of two join attributes has to be calculated to derive the overall join selectivity, which requires additional statistics. As a simple heuristic, we take the selectivity of the most selective join condition as overall join selectivity, and show in Section 2.7 that this results in competitive cardinality estimates. In general, it is sufficient to estimate the cardinality only once per plan class.

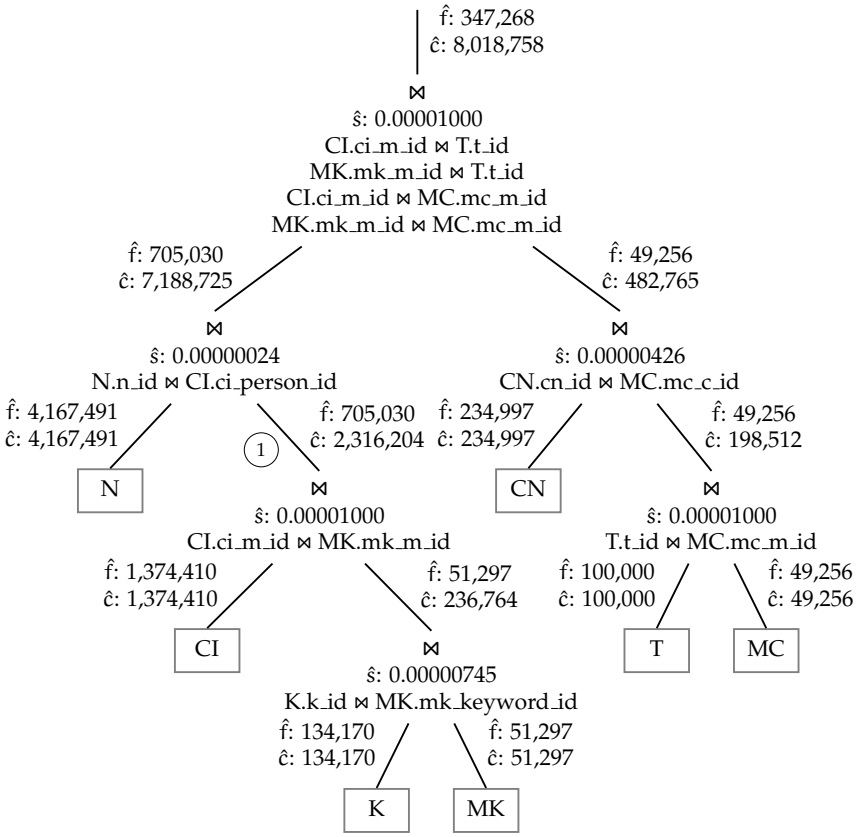


Figure 2.3: Example query execution plan of Join Order Benchmark Query 17, joining the tables `KEYWORD` (K), `MOVIE_KEYWORD` (MK), `CAST_INFO` (CI), `NAME` (N), `COMPANY_NAME` (CN), `TITLE` (T), and `MOVIE_COMPANIES` (MC).

2.6.4 *Parametric Cost Functions*

A foundational building block for the algorithms in the Chapters 3 and 4 are *Parametric Cost Functions* (PCF) [37, 39]:

Definition 2.1. *A Parametric Cost Function (PCF) is the cost of a query execution plan or sub-plan, modeled as a function of one or multiple cost parameters.*

We use PCFs with cardinality as parameter. Consequently, a PCF shows the cost change of a query execution plan or sub-plan, when the cardinality of a base table or intermediate result varies. This is helpful to analyze the impact of cardinality estimation errors, i.e., a difference between estimated and true cardinality.

Next, we give an example for the calculation of a PCF. We consider a robust query execution plan P_{rob} for Query 17 of the Join Order Benchmark [78]. Figure 2.3 shows the query execution plan together with the estimated cardinalities $\hat{f}$ and estimated join selectivities $\hat{s}$. For the the given statistics in Figure 2.3 and C_{out} as cost function,

the entire plan has estimated cost $\hat{c}$ of $8,018,758$. In this example, we create a PCF as a function of the cardinality on edge 1, which could be used to analyze the cost behavior of the query execution plan when cardinality on edge 1 varies. We denote the cardinality on edge 1 as $f_{CI,K,MK}$, since it is the cardinality after joining `CAST_INFO` (CI), `KEYWORD` (K), and `MOVIE_KEYWORD` (MK). To create the PCF, we calculate the C_{out} cost of P_{rob}, but leave $f_{CI,K,MK}$ as variable instead of setting it to the estimated cardinality $\hat{f}_{CI,K,MK} = 705,030$:

$$
\begin{aligned}
&C_{out}(P_{rob}, f_{CI,K,MK}) \\
&= f_{CI,K,MK} + f_{N,CI,K,MK} + f_{N,CI,K,MK,CN,T,MC} \\
&\quad + \hat{f}_{K} + \hat{f}_{MK} + \hat{f}_{K,MK} + \hat{f}_{CI} + \hat{f}_{N} \\
&\quad + \hat{f}_{T} + \hat{f}_{MC} + \hat{f}_{T,MC} + \hat{f}_{CN} + \hat{f}_{CN,T,MC}
\end{aligned}
$$

$$
\begin{aligned}
&f_{N,CI,K,MK} \\
&= f_{CI,K,MK} \cdot \hat{f}_{N} \cdot \hat{s}_{N \bowtie CI,K,MK} \\
&= f_{CI,K,MK} \cdot 4,167,491 \cdot 0.00000024 \\
&= 1.00 \cdot f_{CI,K,MK}
\end{aligned}
$$

$$
\begin{aligned}
&f_{N,CI,K,MK,CN,T,MC} \\
&= \hat{f}_{N,CI,K,MK} \cdot \hat{f}_{CN,T,MC} \cdot \hat{s}_{N,CI,K,MK,CN,T,MC} \\
&= 1.00 \cdot f_{CI,K,MK} \cdot 49,256 \cdot 0.00001 \\
&= 0.49 \cdot f_{CI,K,MK}
\end{aligned}
$$

$$
\begin{aligned}
&C_{out}(P_{rob}, f_{CI,K,MK}) \\
&= f_{CI,K,MK} + 1.00 \cdot f_{CI,K,MK} + 0.49 \cdot f_{CI,K,MK} \\
&\quad + 134,170 + 51,297 + 51,297 + 1,374,410 \\
&\quad + 4,167,491 + 100,000 + 49,256 + 49,256 \\
&\quad + 234,997 + 49,256 \\
&= 2.49 \cdot f_{CI,K,MK} + 6,261,430
\end{aligned}
$$

Except for self-joins, C_{out} always results in PCFs that are linear functions, when modeled as a function of one cost parameter. Linear PCFs simply some calculations we do in this work, such as the slope analysis of PCFs, and the calculation of intersection points between two PCFs. Cost functions other than C_{out} do not necessarily result in linear PCFs. In Chapter 3, we model the costs of different query execution plans as PCF and calculate the intersection point between two PCFs, to identify the cardinality range in which a plan is optimal. In Chapter 4, we also model the costs of different query execution plans as PCF, but take the slope of a PCF as indicator for the robustness of a query execution plan with respect to cardinality estimation errors.

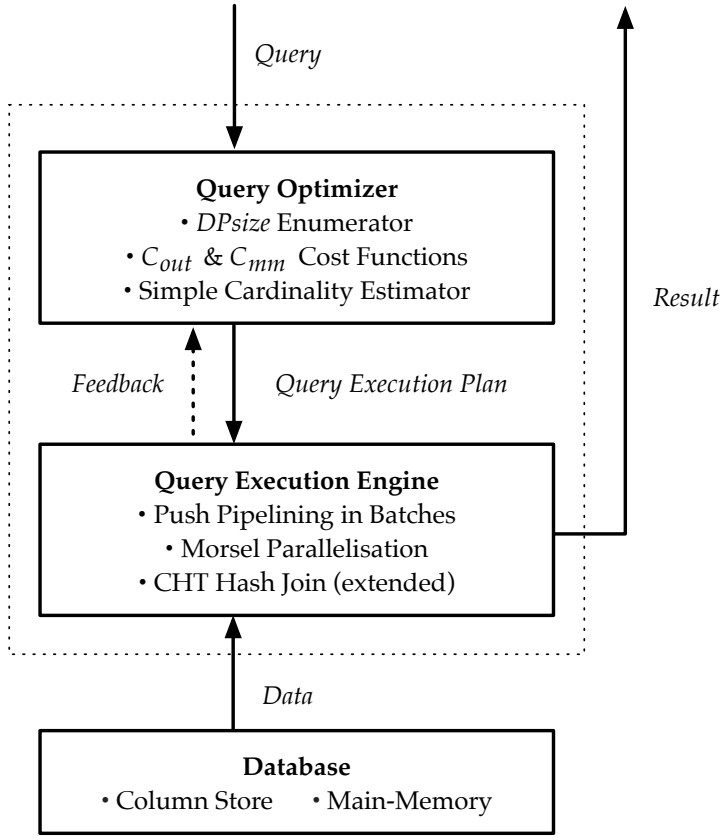


Figure 2.4: Overview of our query processor combining the foundations.

2.7 RESEARCH QUERY PROCESSOR

To implement our approaches and do experimental evaluations, we created a query processor that combines the foundations we discussed in this chapter. Figure 2.4 gives an overview of the basic architecture of our query processor. Our query processor focuses on join processing, and is built on a main-memory column store. The query execution engine implements push-based pipelining in batches and has an extended version of the CHT hash join, which we describe in Section 5.4.2. To better understand the extension of the CHT hash join, we further explain the main-memory column store and execution engine in the Sections 2.7.1 and 2.7.2. For the parallel execution of pipelines, we use morsel-parallelism, including a NUMA-aware task scheduler. On top of the query execution engine, we build a join optimizer, which does dynamic programming enumeration, such as DB2 [17] and Postgres [78]. As Postgres, our join optimizer exhaustively searches the plan space and also enumerates bushy query execution plans. We choose DPsize [4] as dynamic programming enumeration algorithm. Furthermore, our join optimizer uses the C_{out} [26] or C_{mm} [93] cost function, and the simple join cardinality estimator of Section 2.6.3. The query optimizer implementation is single-threaded.

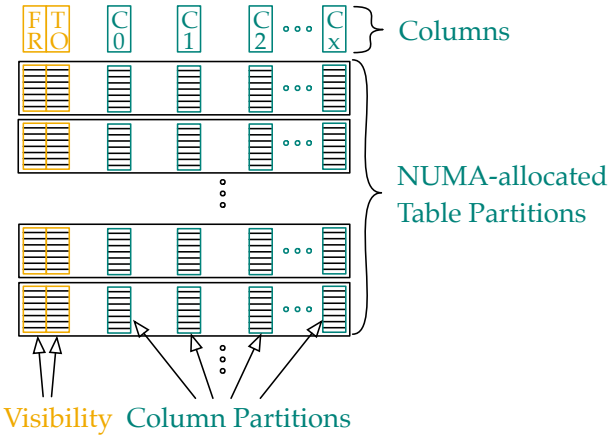


Figure 2.5: Schema showing the table format in our main-memory column store, consisting of columns, NUMA-allocated table partitions and column partitions, and tuple visibility information.

2.7.1 *Main-Memory Column Store*

Figure 2.5 shows the table format of our main-memory column store (see Sections 2.3 and 2.4). Each Table is vertically partitioned into columns, and horizontally partitioned into table partitions. Consequently, the smallest granularity of main-memory that is allocated is a column partition. All table partitions have a fixed size, i.e., a fixed number of tuples. Furthermore, each table partition is allocated on a NUMA-node (see Section 2.3). To consider the effects of parallel inserts, updates, and deletes on the database, such as in HTAP, we added the necessary data structures for *Multi Version Concurrency Control* [5]. There are two additional columns in each table, to store for each tuple the version when is starts being visible and the version when it stops being visible.

2.7.2 *Pipelining*

Our query processor implements push-based pipelining in batches. Figure 2.6 shows an example pipeline, which starts on a base table, as explained in Section 2.7.1, and ends in a pipeline breaker. Pipeline breakers are either the build side of a hash join or the final breaker that materializes the query result. Each pipeline starts on a base table with the creation of batches. Creating a batch works as follows:

We iterate over the table partitions in the table. For each table partition we create a task, which is assigned to the same NUMA-node on which the table partition is allocated. Next we submit the task to our NUMA-aware task scheduler. Each NUMA-task creates a batch based on the corresponding table partition. Since table partitions and batches have the same size, i.e., the same number of tuples, and all data structures are allocated in main-memory, we just add references

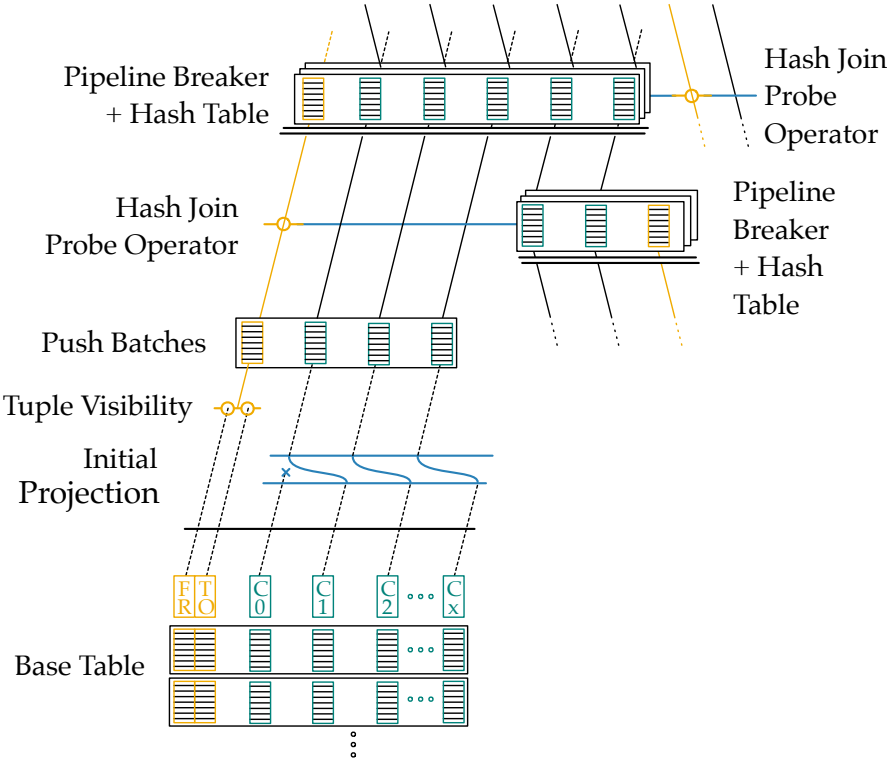


Figure 2.6: Schema of the pipelining in our query execution engine, showing the creation of batches, join operators and pipeline breakers.

of the required column partitions of the table partition to the batch. This avoids copying data from the database into the query processor. Each batch has furthermore a bitset to indicate which of its tuples are valid. For instance, not finding a partner in an equi-join can make a tuple invalid. Initially the bitset is filled by evaluating the Multi Version Concurrency Control information. Each query reads exactly one version of the database, so we initially check which tuples are valid in that version, and store this information in the bitset.

Next, each NUMA-task pushes its batch through the pipeline until it has no tuples anymore, or reaches the pipeline breaker. Each time a batch is pushed from one operator to another creates a virtual function call. This first operator could be the probe side of a hash join, as in Figure 2.6. To figure out which tuples are still valid, each operator initially checks the tuple validity bitset, which is small enough to reside in the processor cache. Each join operator can add temporary column partitions to the batch to store the tuples of the joined table. Since allocating and deallocating temporary column partitions can introduce considerable overheads, we employ a corresponding pool allocator. Joins can also be $m{:}n$ joins, so that one tuple on the probe side in an equi-join finds multiple join partners on the build side. In this case the first join partner is materialized in the actual batch, and all other join partners are materialized together with the original probe-side tuple into an overflow batch. Tuples that do not find a join

partner are marked as invalid, so that they are not considered in the the following operators in the pipeline. For each full overflow batch, a new NUMA-task is created, which pushes the overflow batch through the remaining operators in the pipeline to the pipeline breaker. In the end, all batches that contain valid tuples are collected in the pipeline breaker, and the NUMA-tasks finish.

2.7.3 *Experimental Verification*

To verify our query processor design, we repeated some experiments from Leis et al. [78]. First, we show that the C_{out} and C_{mm} cost functions correlate to the execution times in our query execution engine. We executed different query execution plans for the queries of the Join Order Benchmark [78]. The Figures 2.7 and 2.8 show the results for C_{out} and C_{mm}. The x-axis shows the costs based on true cardinalities, and the y-axis the query execution times. Similar to the experiments in Leis et al. [78], there is a strong correlation between the true cost and the query execution time. Consequently, it is reasonable to choose C_{out} and C_{mm} as cost functions for our query optimizer.

Figure 2.9 presents the results of the second experiment, which shows the errors in the estimation of join cardinalities. The x-axis shows the number of joins, and the y-axis the factor of estimation error. Similar to the cardinality estimators, which were compared in Leis et al. [78], our cardinality estimator underestimates the join cardinalities. Also the numbers are similar to the results in Leis et al. [78]: For six joins, our estimator has a median estimation error of two orders of magnitude, and at most a four orders of magnitude.

Our experiments confirm the result of Leis et al. [78]. We see that precise cardinality estimation is rather an issue than a good cost function. Furthermore, our experiments show that our cardinality estimator and cost function have the same behavior for the Join Order Benchmark, as the state-of-the art systems compared in Leis et al. [78]. In the end, the experiments verify the design of our query processor.

Our query processor was finalist in the *ACM SIGMOD Programming Contest 2018*. Due to the dynamic programming enumeration algorithm we choose, the bushy trees we enumerate, and the results of the experiments in the Figures 2.7, 2.8, and 2.9, we argue that our join optimizer's choice of the estimated optimal plan is very similar to the choice of popular commercial and free systems, for the considered benchmark. We use this query processor to implement our approaches and do experiments. The optimality range calculation in Chapter 3, and the robustness metrics and the robust plan selection in Chapter 4 are implemented as extensions of the query optimizer. For the adaptive query processor in Chapter 5, we extend both the query optimizer and the query execution engine.

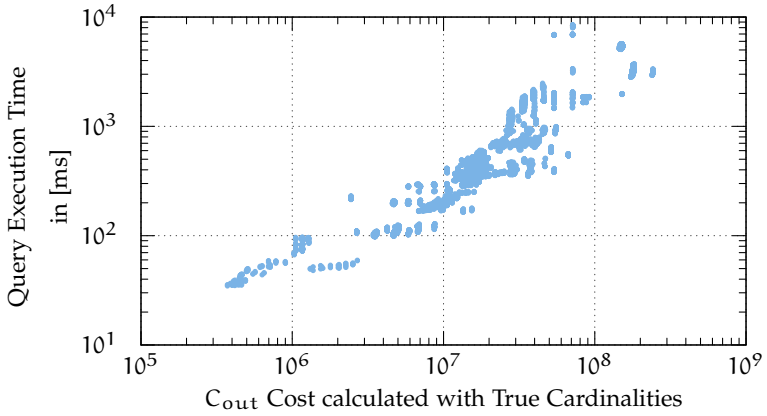


Figure 2.7: Experiment from Leis et al. [78] repeated for our query processor to show the correlation between the C_{out} cost function and query execution times in our query execution engine.

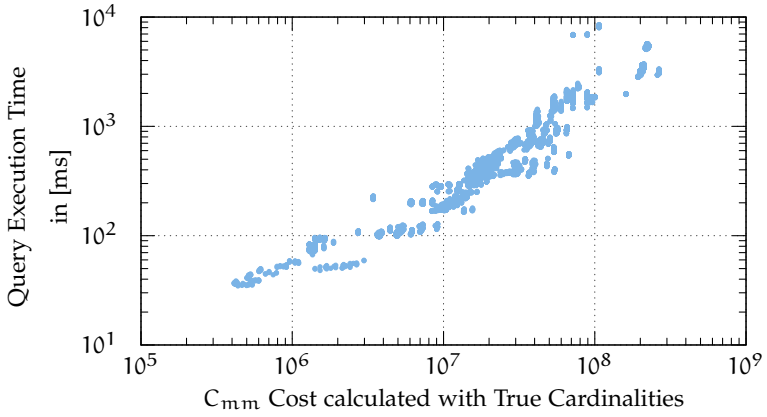


Figure 2.8: Experiment from Leis et al. [78] repeated for our query processor to show the correlation between the C_{mm} cost function and query execution times in our query execution engine.

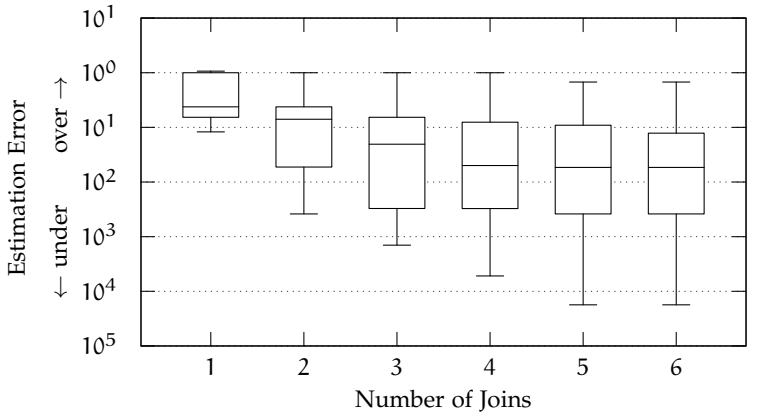


Figure 2.9: Experiment from Leis et al. [78] repeated for our cardinality estimator to show the estimation errors after some joins.

3

PRECISE OPTIMALITY RANGES

The first problem, we address in this work is the detection of sub-optimal query execution plans, caused by cardinality estimation errors. Our solution are precise optimality ranges, which cannot only detect sub-optimal query execution plans, but have further applications in query processing.

3.1 INTRODUCTION

Intermediate result cardinalities are the central parameters of each cost model, and consequently have a strong impact on the choice of the optimal plan. In this chapter we look into the following problem: A query optimizer selects a query execution plan that it considers to be optimal. We assume that in the optimal plan, the cardinality of an intermediate result I is not the estimated cardinality $\hat{f}_I$ but variable, denoted as f_I. This covers situations in which the true cardinality $\mathring{f}_I$ is different to the estimated cardinality $\hat{f}_I$. It can occur because of estimation errors, because it was a parametric query, or because of database changes in the meantime as in HTAP. The emerging question is: For which cardinalities does the selected optimal plan remain optimal? It can be answered by an optimality range for cardinalities $r_I = [f_I^{\downarrow}; f_I^{\uparrow}]$, so that $0 \leqslant f_I^{\downarrow} \leqslant \hat{f}_I \leqslant f_I^{\uparrow} \leqslant \infty$. Optimality ranges can be annotated on each edge in the query execution plan, i.e., for each intermediate result. As soon as a part of the query or the entire query was executed, the true cardinalities for the intermediate results are available. If a true cardinality falls outside of its pre-calculated optimality range, we know there is another, cheaper plan. This simple check is much more efficient than a full optimization.

Figure 3.1 shows a running example for this chapter: It is a query and the its optimal query execution plan with annotated optimality ranges. The query is a chain query joining five relations with estimated cardinalities $\hat{f}$ and estimated join selectivities $\hat{s}$ as depicted on the top of Figure 3.1. In the middle of Figure 3.1 we see the cost-optimal plan P_{opt} with annotated optimality ranges r. Note that some lower bounds are zero and some upper bounds are infinity. One reason is that the number of alternative plans decreases the more subplans get executed. Suppose the join of R and S in the cost optimal plan P_{opt} was executed. The estimated cardinality of $R \bowtie S$, $\hat{f}_{R \bowtie S}$ is 10^4, and the optimality range on the outgoing edge of $R \bowtie S$, $r_{R \bowtie S}$ indicates that the plan remains optimal when the cardinality $f_{R \bowtie S}$ is between 10^3 and 10^5. Note that optimality range bounds are of course

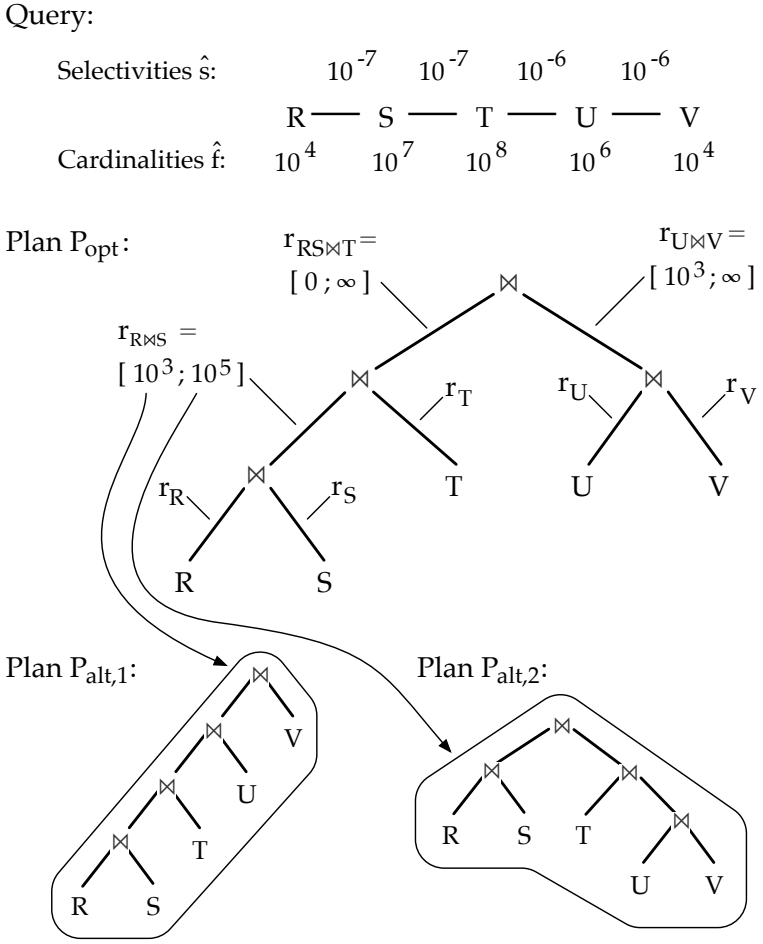


Figure 3.1: Example for an optimal query execution plan with optimality ranges on its edges to indicate where the plan is optimal.

not always decimal powers. When the cardinality $f_{R \bowtie S}$ exceeds the upper bound $f^{\uparrow}_{R \bowtie S} = 10^5$, the alternative plan P_{alt2} becomes optimal.

Calculating precise optimality ranges is, to the best of our knowledge, an unsolved problem. We present an algorithm to calculate precise optimality ranges, which relies on cost-based query optimization. Our solution contains plan space and cost parameter considerations. The algorithm can be a post-optimization step, and utilize the work and data structures from the regular plan search.

Next to the algorithm to calculate precise optimality ranges, this chapter has the following contributions: Together with the algorithm, Section 3.2 presents the corresponding building blocks, the pruning strategy, and a worst case analysis for the number of enumerated plans to calculate precise optimality ranges. Additional contributions are the experimental analysis on optimality ranges for a TPC-H-like benchmark, and an experimental evaluation on the number of enumerated plans during optimality range calculation. Both are presented in Section 3.3. We finish this chapter with discussing related work and applications for optimality ranges in the Sections 3.4 and 3.5.

3.2 CALCULATION OF PRECISE OPTIMALITY RANGES

We describe the calculation of a precise optimality range for an edge in an optimal query execution plan. Our optimality range calculation is based on the plan table, e.g. the MEMO structure or dynamic programming table, which was created during the regular query optimization. A plan table consists of groups or plan classes. Every group or plan class represents a set of plans with the same logical properties, e.g. the same set of referenced tables. The objective of optimality ranges is to indicate for which intermediate result cardinalities the optimizer would not find a better plan. For that reason, the calculation takes the same inputs and makes the same assumptions as the optimizer. Furthermore, this decision is consistent when the enumeration for initial optimization and optimality range calculation is deterministic, i.e., the same plan space is enumerated. Optimality range calculation works for optimization algorithms that guarantee to find the cost-optimal plan, such as bottom-up [14, 54] and top-down [46, 67] dynamic programming, or transformation-based [19, 29] optimization. It also works for optimization algorithms that enumerate a subset of the necessary plan space, when the enumeration is deterministic. The meaning of optimality ranges for randomized and genetic optimization algorithms is questionable: They could indicate a better plan which is not found when the optimizer is invoked again.

In Section 3.2.3, we discuss the plan alternatives, which are the feasible plans [14] or also denoted as connected-sub-graph complement pairs [44] enumerated by dynamic programming. Just as dynamic programming, our optimality range calculation algorithm (Section 3.2.4) exploits Bellman's principle of optimality [57]. We argue that this is the most efficient way to calculate optimality ranges. We also explain how optimality ranges are calculated with other enumeration strategies. As a general procedure for optimality range calculation, we propose to search the optimal plan, keep the plan table, and then calculate the optimality ranges for the optimal plan. For the next subsections we make the following assumptions:

1. We assume query execution plans contain only scan and join operators.
2. Plan classes keep only one optimal plan, and no further plans with additional physical properties like sorted columns.
3. We assume a cost function that results in linear Parametric Cost Functions (PCFs).
4. There are no estimation errors for parallel sub-plans in bushy query execution plans.

Since these assumptions do not hold in reality, we discuss in Section 3.2.6 how they can be relaxed.

3.2.1 *Plan Cost Intersection*

The first building block for optimality range calculation is the intersection of query execution plan costs [37, 39]. It starts with modeling the cost of the optimal plan as PCF (see Section 2.6.4) of the cardinality of the edge on which the optimality range is calculated. This is repeated for a corresponding alternative plan. Next, the intersection point of the two PCFs is calculated to figure out at which cardinality the alternative plan becomes cheaper, i.e., the optimal plan stops being optimal. The necessary plan alternatives, which have to be intersected with the optimal plan to calculate a precise optimality range are explained in Section 3.2.3.

We continue with an example for cost intersection. To simplify the example calculations, we use the C_{out} cost function, but set the cost of base table scans to zero, so that:

$$C_{out}(P) = \begin{cases} 0 & \text{if } P \text{ is a base table } R \\ |P| + C_{out}(P_1) + C_{out}(P_2) & \text{if } P = P_1 \bowtie P_2 \end{cases} \tag{3.1}$$

The C_{out} cost function is no strong limitation, and our approach also works with cost functions other than C_{out}. Let us consider the example query in Figure 3.1 for the cost intersection again. For C_{out} and the given statistics, plan P_{opt} has costs of $1.21 * 10^5$, and plan P_{alt2} has costs of $1.021 * 10^6$. Consequently, P_{opt} was preferred over plan P_{alt2}. Now we assume cardinality changes on the $R \bowtie S$ edge, and derive the corresponding PCF for P_{opt}. We calculate the cost of P_{opt} and assume that the output cardinality of $R \bowtie S$, $f_{R\bowtie S}$ or f_{RS} is not the estimated cardinality $\hat{f}_{RS} = 10^4$ but an arbitrary value:

$$C_{out}(P_{opt}, f_{RS}) = f_{RS} + \hat{f}_{RST} + \hat{f}_{UV} + \hat{f}_{RSTUV} \tag{3.2}$$
$$\hat{f}_{RST} = f_{RS} * 10^8 * 10^{-7} = 10 * f_{RS} \tag{3.3}$$
$$\hat{f}_{UV} = 10^6 * 10^4 * 10^{-6} = 10^4 \tag{3.4}$$
$$\hat{f}_{RSTUV} = f_{RST} * f_{UV} * 10^{-6} = 0.1 * f_{RS} \tag{3.5}$$
$$C_{out}(P_{opt}, f_{RS}) = 11.1 * f_{RS} + 10^4. \tag{3.6}$$

We also derive the PCF for P_{alt2}:

$$C_{out}(P_{alt2}, f_{RS}) = \hat{f}_{UV} + \hat{f}_{TUV} + f_{RS} + \hat{f}_{RSTUV} \tag{3.7}$$
$$\hat{f}_{UV} = 10^6 * 10^4 * 10^{-6} = 10^4 \tag{3.8}$$
$$\hat{f}_{TUV} = 10^8 * f_{UV} * 10^{-6} = 10^6 \tag{3.9}$$
$$\hat{f}_{RSTUV} = f_{RS} * f_{TUV} * 10^{-7} = 0.1 * f_{RS} \tag{3.10}$$
$$C_{out}(P_{alt2}, f_{RS}) = 1.1 * f_{RS} + 1.01 * 10^6. \tag{3.11}$$

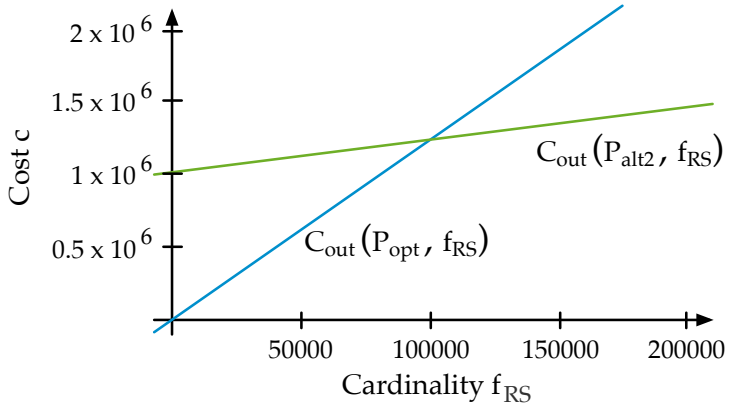


Figure 3.2: Example for the intersection of the cost of the optimal query execution plan and an alternative query execution plan.

Due to the characteristics of C_{out}, we get PCFs for $C_{out}(P_{opt}, f_{RS})$ and $C_{out}(P_{alt2}, f_{RS})$ that are linear functions, and consequently it is trivial to calculate their intersection point:

$$C_{out}(P_{opt}, f_{RS}) = C_{out}(P_{alt2}, f_{RS}) \tag{3.12}$$

$$11.1 * f_{RS} + 10^4 = 1.1 * f_{RS} + 1.01 * 10^6 \tag{3.13}$$

$$f_{RS} = 10^5. \tag{3.14}$$

Comparing the slopes of $C_{out}(P_{opt}, f_{RS})$ and $C_{out}(P_{alt2}, f_{RS})$ (see Figure 3.2) shows that P_{opt} is cheaper when f_{RS} is smaller than 10^5, and P_{alt2} otherwise. Two PCFs can also be parallel, i.e., no intersection point, or they intersect in the negative range. For cost functions other than C_{out}, there can be non-linear PCFs, which require more effort to calculate one or potentially multiple intersection points between two PCFs [39]. In the end, modeling the costs of plans as PCF and calculating the intersection point of two PCFs is the first building block for the optimality range calculation algorithm.

We utilize this procedure in *Narrow Range*, which we describe in Algorithm 3.1: Given the PCF of an optimal query execution plan pcf_{opt}, the PCF of another query execution plan pcf_{alt}, and an optimality range, Algorithm 3.1 potentially returns an optimality range with more narrow bounds. It starts by calculating the intersection point of pcf_{opt} with pcf_{alt} (Line 3). If the intersection point is out of the current optimality range (Line 4), then the optimality range could not be narrowed and is returned unmodified (Line 5). If the intersection point is in the current optimality range, we can narrow the optimality range. In this case, we have to check if the intersection point restricts the optimality range on the lower or upper end, i.e., it is a new lower or upper bound (Line 7). In both cases, we return a more narrow optimality range (Lines 8 and 10).

Algorithm 3.1 Narrow Range

```
1: function narrowRange(PCF_ax+b pcf_opt, PCF_ax+b pcf_alt, Cardinality f_I^↓, Cardinality f_I^↑)
2:     declare Cardinality f_I^intersect
3:     f_I^intersect ← (pcf_opt.b − pcf_alt.b)/(pcf_alt.a − pcf_opt.a)
4:     if f_I^intersect < f_I^↓ or f_I^intersect > f_I^↑ then
5:         return ⟨f_I^↓, f_I^↑⟩
6:     end if
7:     if pcf_opt.a < pcf_alt.a then
8:         return ⟨f_I^intersect, f_I^↑⟩
9:     else
10:        return ⟨f_I^↓, f_I^intersect⟩
11:    end if
12: end function
```

3.2.2 *Optimal Plans Container*

Another building block for the optimality range calculation is the *Optimal Plans Container* (OPC):

Definition 3.1. *An Optimal Plans Container (OPC) is a container for PCFs, and has a lower and upper cardinality bound. The lower and upper bound is initialized by the calculation algorithm with the current optimality range. A PCF is only inserted into an OPC, when there is at least one point between the lower and upper bound where the PCF is optimal. The insertion of a PCF can cause the pruning of other PCFs in the container.*

We now describe an OPC for PCFs that are linear functions. Figure 3.3 shows an example OPC for linear PCFs, with the cardinality f_I of an intermediate result I denoted on the x-axis and the plan costs c on the y-axis. The OPC has the lower bound $f_I^{\downarrow}$ and the upper bound $f_I^{\uparrow}$, and contains three PCFs that are piecewise optimal in the range. Furthermore, Figure 3.3 shows the PCF_0, which was in the OPC until it got pruned during the insertion of PCF_3.

In Algorithm 3.2, we show how to insert a linear PCF pcf_{new} into an OPC opc. An OPC stores each PCF together with the cardinality value from where its plan starts to be optimal. In Figure 3.3: PCF_1 is optimal from $f_I^{\downarrow}$, PCF_3 from f_I', and PCF_2 from f_I''. An OPC is a list of tuples, composed of a cardinality value and a PCF (Lines 2 and 3). The list is sorted by the cardinality values. Furthermore, an OPC contains variables for its lower and upper bound, $f_I^{\downarrow}$ and $f_I^{\uparrow}$ (e.g. Line 26).

A PCF pcf_{new} is only inserted into pcfs when there is at least one point between $f_I^{\downarrow}$ and $f_I^{\uparrow}$ where it is cheaper than all PCFs in pcfs. Since we have linear PCFs, it is sufficient to only check the corners, i.e., the cardinality points f_I. So initially we run over all ordered tuples of cardinality values f_I and PCF pcf in pcfs (Line 11). If there is one PCF that is equal to pcf_{new}, we immediately return without inserting

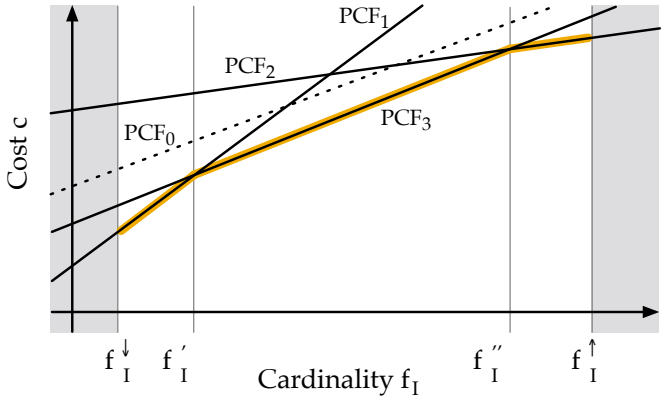


Figure 3.3: Example Optimal Plans Container (OPC) with its lower and upper bound, containing three PCFs.

pcf_{new}(Lines 12 to 14). If there is no such case, we store the first and the last cardinality value f_I where $pcf_{new}(f_I)$ is smaller than $pcf(f_I)$ in f_I^{first} and f_I^{last} (Lines 15 to 20). After the loop we determine if (Line 22) and where to insert pcf_{new} (Lines 25 to 30), update the key of the following entry in $pcfs$ (Line 31), and potentially delete PCFs that got sub-optimal because of pcf_{new} (Line 32). Finally, we insert pcf_{new} with its key at the corresponding position in $pcfs$ (Line 33).

The pruning of PCFs during the insertion into an OPC significantly reduces the number of enumerated plans during optimality range calculation. We further explain the impact of pruning in Section 3.2.4.2. Pruning can also be disabled by just inserting PCFs into the container (Lines 4 to 7). We show the significant difference between enabled and disabled pruning in Section 3.3.2.

3.2.3 *Considered Plan Alternatives*

In this section we consider the plan space to decide which plans have to be compared when calculating the optimality range for a given edge in an optimal query execution plan. The central idea behind dynamic programming is Bellman's principle of optimality [57], i.e., an optimal solution can always be constructed from optimal sub solutions. To find the optimal query execution plan, dynamic programming query optimizers [4] usually do not keep only one optimal sub-plan, but additional sub-plans that have certain properties such as sorted columns. For now, we ignore physical properties to keep things simple, and assume that only the optimal sub-plan is kept. Figure 3.4 shows the dynamic programming plan space for the example query in Figure 3.1. The boxes denote different plan classes, which are a set of alternative plans that represent the same intermediate result. The plan class RST for instance contains the two alternative plans R, ST and RS, T. All plan classes are stored in the dynamic programming

Algorithm 3.2 Optimal Plans Container Insertion

```
1: function INSERTINTOOPC(OPC opc, PCF pcf_new, bool pruning)
2:     declare SortedList⟨Cardinality, PCF⟩ pcfs
3:     pcfs ← opc.pcfs
4:     if !pruning then
5:         pcfs.insert(0, pcf_new)
6:         return
7:     end if
8:     // run over the entries, check where pcf_new is smaller
9:     declare Cardinality f_I^first ← ∞
10:    declare Cardinality f_I^last ← ∞
11:    for each ⟨Cardinality f_I, PCF pcf⟩ in pcfs do
12:        if pcf = pcf_new then
13:            return
14:        end if
15:        if pcf_new(f_I) ⩽ pcf(f_I) then
16:            if f_I^first = ∞ then
17:                f_I^first ← f_I
18:            end if
19:            f_I^last ← f_I
20:        end if
21:    end for
22:    if f_I^first = ∞ then
23:        return
24:    end if
25:    declare Cardinality f_I^key
26:    if f_I^first = opc.f_I^↓ then
27:        f_I^key ← opc.f_I^↓
28:    else
29:        f_I^key ← intersect(pcf_new, pcfs.at(f_I^first).prev())
30:    end if
31:    pcfs.at(f_I^last).f_I ← intersect(pcf_new, pcfs.at(f_I^last))
32:    pcfs.eraseIncludingBounds(f_I^first, f_I^last)
33:    pcfs.insert(f_I^key, pcf_new)
34:    opc.pcfs ← pcfs
35: end function
```

table, together with their optimal. For cost functions other than C_{out} there can be of course more plan alternatives. Asymmetric cost functions such as C_{mm} consider different build and probe side of hash joins. Consequently more alternative plans can be enumerated. C_{out}, which we use as example, covers only different operator orders, i.e., the logical execution plan. Operator implementation, e.g. hash join vs. sort merge join, and operator input order, i.e., build vs. probe side are not covered by C_{out}. The different plans in a plan class usually

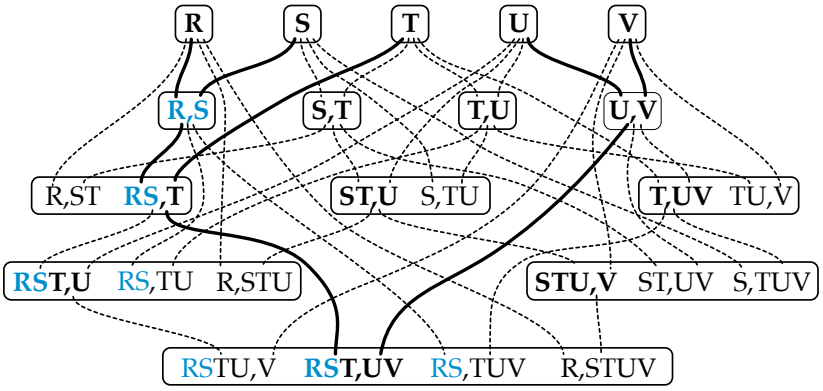


Figure 3.4: Dynamic programming plan space of the example in Figure 3.1 showing the plan classes with their plan alternatives. The optimal plans are shown in bold font.

have different costs, so that only one is optimal (depicted in bold font). In the plan class RST for instance, the plan RS, T is the optimal plan, which is created out of the optimal plans of the plan classes RS and T. We only store one optimal plan per plan class, and prune more expensive plans (depicted in regular font). Figure 3.4 also shows the plan P_{opt} of Figure 3.1 in solid lines.

Definition 3.2. *A Dependence Plan Class (DPC) is a plan class, and always part of the optimal plan. On its outgoing edge in the optimal plan, the optimality range is calculated. Furthermore, its output cardinality is a parameter of the PCFs, which are created for the optimality range calculation.*

Let us now assume we calculate the optimality range of P_{opt} (solid lines) on the outgoing edge of $R \bowtie S$. We call $R \bowtie S$ or RS the DPC (Definition 3.2). The relevant plan classes to consider are the ones that are directly or indirectly referencing RS, i.e., the ones where a change of $f_{R \bowtie S}$ has an impact on the optimal plan choice. For P_{opt} and RS as DPC, those are the plan classes RS, RST, RSTU, and RSTUV. The other plan classes are independent of $f_{R \bowtie S}$, and have consequently constant costs with respect to $f_{R \bowtie S}$. In general, all plan classes that are directly or indirectly referencing the DPC have to be considered.

To compare the costs of different plans, they have to represent the same intermediate result. So comparing the costs can only be done within the same plan class. Since conventional dynamic programming optimization stores only one optimal plan per plan class, we have to either modify it to store all enumerated alternatives, or re-enumerate plans during the optimality range calculation. We re-enumerate the plan alternatives, since not all plan classes are considered during the optimality range calculation. Furthermore, storing all plan alternatives for each plan class can increase the memory consumption and initial optimization time. In the plan classes that directly or indirectly reference the DPC, we distinguish three different types of plans:

1. The optimal plan, which by definition directly or indirectly references the DPC. An example is the plan RST, U in the plan class $RSTU$, considering P_{opt} (solid lines in Figure 3.4) with RS as DPC.

2. Plans that are not optimal, and directly or indirectly reference the DPC, such as the plan RS, TU in the plan class $RSTU$.

3. Plans that do not reference the DPC, such as the plan R, STU in the plan class $RSTU$. For the latter type of plan, the costs are constant and therefore the PCF parallel to the x-axis.

3.2.4 *Calculation Algorithm*

In this section we describe the algorithm to calculate the optimality range for one edge in a given optimal plan. It is a recursive algorithm, which outputs a sharp lower and upper bound for the optimality of the plan. Initially, the lower and upper bound of the optimality range are 0 and ∞, respectively. The basic idea is to compare sub-plans of the given optimal execution plan with its corresponding alternatives and incrementally narrow the optimality range. By comparing sub-plans of the optimal plan with its alternatives, we utilize Bellman's principle of optimality. We argue that this is the most efficient way, since the ranges converge faster, which at the same time enables more effective pruning. Nevertheless, optimality range calculation works with any deterministic enumeration strategy, because we can also compare the entire optimal plan with corresponding plan alternatives. For the correctness, it does not matter how we enumerate the plans that are compared against the optimal plan. We start with explaining optimality range calculation using the example query of Figure 3.1 in Section 3.2.4.1, discuss the pruning in Section 3.2.4.2, and present the final algorithm in Section 3.2.4.3.

3.2.4.1 *Example*

We use the example query and statistics of Figure 3.1 with the corresponding dynamic programming table in Figure 3.4. The optimal plan for the example query is P_{opt}, which is depicted in Figure 3.4 in solid lines. Furthermore, we choose the plan class RS as DPC, i.e., we want to calculate the optimality range on the outgoing edge of $R \bowtie S$. So the first promising comparison is in the plan class RST, where we start with re-enumerating all sub-optimal plan alternatives. In the plan class RST, there is the plan RS, T as part of P_{opt}, and the plan R, ST that does not reference RS. The question to answer now is: For which values of $f_{R \bowtie S}$ has R, ST smaller cost than RS, T? So we model the costs of both plans as PCF with RS as DPC and calculate the intersection point as described in Section 3.2.1. Remember that

the plan R, ST does not reference RS and therefore has constant cost. The PCF of R, ST (see Section 3.2.1):

$$C_{out}(RS \bowtie T, f_{R \bowtie S}) = f_{R \bowtie S} + \hat{f}_{RS \bowtie T} = 11 * f_{R \bowtie S} \tag{3.15}$$

intersects with the PCF of RS, T:

$$C_{out}(R \bowtie ST, f_{R \bowtie S}) = 10,010,000 \tag{3.16}$$

at $f_{R \bowtie S} = 9.1 * 10^6$. Since this point is inside our current optimality range $r_{R \bowtie S} = [0; \infty]$, we get a new lower or upper bound. Algorithm 3.1 reveals that the plan RS, T is cheaper until $f_{R \bowtie S} = 9.1 * 10^6$, and plan R, ST beyond this point. We just considered a sub-plan of the optimal plan. Since Bellman's principle of optimality holds, we know that the optimal plan is not optimal anymore when $f_{R \bowtie S}$ becomes larger than $9.1 * 10^6$. Consequently, we narrow the optimality range to $r_{R \bowtie S} = [0; 9.1 * 10^6]$. With the next comparisons, we are going to further narrow the optimality range. Note that intersection points beyond our current optimality range are irrelevant, since they cannot further narrow the optimality range.

The next interesting plan comparisons are in the final plan class RSTUV, for which we also re-enumerate all non-optimal plan alternatives first. Next, we compare the plan RST, UV as part of P_{opt} with its alternatives. The first alternative is the plan RS, TUV, which is not part of the optimal plan but references the DPC. We model the costs of RS, TUV as PCF:

$$C_{out}(RS \bowtie TUV, f_{R \bowtie S}) = 1.1 * f_{R \bowtie S} + 1,010,000 \tag{3.17}$$

and the costs of our optimal plan RST, UV as PCF:

$$C_{out}(RST \bowtie UV, f_{R \bowtie S}) = 11.1 * f_{R \bowtie S} + 10,000 \tag{3.18}$$

to calculate their intersection point. As a result, we get a new upper bound and update our optimality range to $r_{R \bowtie S} = [0; 10^5]$ (see Algorithm 3.1). Another alternative plan is R, STUV, which does not reference the DPC RS, and therefore has constant cost. We also model the cost of plan R, STUV as PCF:

$$C_{out}(R \bowtie STUV, f_{R \bowtie S}) = 2,011,000 \tag{3.19}$$

and calculate the intersection point with the PCF of our optimal plan RST, UV (Equation 3.18). Since we get an intersection point that is beyond our current upper bound, we ignore it (see Algorithm 3.1).

The most interesting plan to compare is RSTU, V, since there are multiple alternatives for the plan class RSTU, and some plans directly or indirectly reference the DPC RS: The first two alternatives are the plans R, STU and RS, TU. Two additional alternatives result from the plan RST, U, since the plan class RST also references the DPC RS. In

sum, there are four different plan alternatives for the plan $RSTU, V$ that have to be compared with the optimal plan RST, UV. Note that dynamic programming would not enumerate all four alternatives because of Bellman's principle of optimality. But in order to get the exact optimality range, we have to consider these alternatives. Only one of the four alternatives:

$$C_{out}(RSTU \bowtie V, f_{R \bowtie S}) = 21.1 * f_{R \bowtie S} \tag{3.20}$$

intersects with the PCF of P_{opt} (Equation 3.18) in the current optimality range $r_{R \bowtie S} = [0; 10^5]$, which gives us a new lower bound $f^{\downarrow}_{R \bowtie S} = 10^3$ (see Algorithm 3.1). Now that all the relevant plans were considered, we have the final optimality range $r_{R \bowtie S} = [10^3; 10^5]$ for the outgoing edge of $R \bowtie S$ in plan P_{opt} as depicted in Figure 3.1.

3.2.4.2 *Pruning*

When comparing plan alternatives with an optimal plan, it is in practice not necessary to enumerate all possible plans of a plan class, as we did for the plan class $RSTU$ in the example in Section 3.2.4.1. To calculate correct optimality ranges, it is sufficient to enumerate only plans that consist of sub-plans that are somewhere optimal, due to Bellman's principle of optimality. We can further limit that to sub-plans that are optimal somewhere within the current optimality range, since we are not interested in intersection points beyond the current optimality range. This gives us a powerful pruning strategy, which significantly reduces the number of enumerated plans (see Section 3.3.2). To explain the pruning, we consider the example in Section 3.2.4.1. The pruning strategy removes three of the four alternatives for the plan $RSTU, V$.

We utilize the Optimal Plans Container (OPC) presented in Section 3.2.2. The first OPC is created for the plan class $RSTU$ when the plan $RSTU, V$ is supposed to be compared with the optimal plan RST, UV. The OPC for $RSTU$ is initialized with the current optimality range $r_{R \bowtie S} = [0; 10^5]$, i.e., will only contain PCFs that are piecewise optimal in the optimality range. Before the OPC is filled with the PCFs of plans from the plan class $RSTU$, we recursively invoke this procedure for the plan RST, U, i.e., the plan class RST. This recursion towards the DPC RS stops once the DPC RS is reached. For the plan class RST, we add both plan alternatives to the corresponding OPC (see Algorithm 3.2), while only the plan RS, T is optimal within the current optimality range. This is because the plan class RST is part of the optimal plan, and is already considered in the current optimality range. In the end, this removes the first of the four alternatives for the plan $RSTU, V$. When the recursion returns from plan class RST to plan class $RSTU$, we propagate all plans from the OPC of RST, i.e., only one plan, to $RSTU$, and insert it in the OPC for $RSTU$ (Algorithm 3.2). Furthermore, we add the plans RS, TU, and R, STU to the

Algorithm 3.3 Start Optimality Range Calculation

```
1: function startOptimalityRangeCalc(PlanTable pt, DPC dpc)
2:     // start the recursion on the final plan class
3:     declare PlanClass finalPc ← pt.finalPlanClass
4:     declare Cardinality f_I^↓
5:     declare Cardinality f_I^↑
6:     declare PCF pcf_opt
7:     ⟨f_I^↓, f_I^↑, pcf_opt⟩ ← calculateOptimalityRange(finalPc, dpc)
8:     return ⟨f_I^↓, f_I^↑⟩
9: end function
```

OPC for RSTU. Since they are at no point optimal within the current optimality range, they are ignored (see Algorithm 3.2). This prunes another two of the four alternatives for the plan RSTU, V. In the end, there is only one plan for RSTU, V that has to be compared with the optimal plan RST, UV.

3.2.4.3 *Formal Algorithm Description*

We formalized the optimality range calculation in the Algorithms 3.3, 3.4, and 3.5. Algorithm 3.4 compares the optimal plan parts with their alternatives to narrow the optimality range as described in Section 3.2.4.1. Algorithm 3.5 searches all necessary plan alternatives and implements the pruning strategy of Section 3.2.4.2. We expect a fully optimized plan and the plan table still populated with one optimal plan per plan class. As query execution plan format we assume plan nodes that only know their children, so that a plan is just a pointer to the top plan node. Optimality range calculation is a recursive algorithm, which starts on the final plan class, and traverses the path in the optimal plan to the DPC, which is the recursion base. In the example in Figure 3.4, the recursion starts on the plan class RSTUV, continues on plan class RST, and ends in the DPC RS. Algorithm 3.3 only starts the recursion by invoking Algorithm 3.4 (Line 7) on the final plan class (Line 3), and returns the final optimality range (Line 8).

Algorithm 3.4 is the main recursive function that is invoked for every plan class on the path between the final plan class and the DPC. Next to the considered plan class pc, it also takes the DPC as parameter (Line 1). In the recursion base, i.e., pc being equal to dpc, it returns the initial optimality range 0 to ∞ and a PCF for the DPC, which is an identity function plus the constant cost of the subplans (Lines 3 to 5). In case we are not in the recursion base, we first do a recursive invocation to the next plan class on the path to the DPC (Lines 8 to 13). Returned is the current optimality range and the PCF for the sub-plan of the optimal plan (Line 13), which is first propagated to the current plan class (Line 15) and then used for a comparison with other plan alternatives in this plan class. So in the

Algorithm 3.4 Calculate Optimality Range

```
1:  function calculateOptimalityRange(PlanClass pc, DPC dpc)
2:      // recursion base
3:      if pc = dpc then
4:          return ⟨0, ∞, PCF(f_I) = f_I + c_const⟩
5:      end if
6:      // get next plan class towards the DPC
7:      declare PlanClass nextPc
8:      nextPc ← getSubPlanToDpc(pc.optPlan, dpc)
9:      // recursive invocation, traverse to nextPc
10:     declare Cardinality f_I^↓
11:     declare Cardinality f_I^↑
12:     declare PCF pcf_opt
13:     ⟨f_I^↓, f_I^↑, pcf_opt⟩ ← calculateOptimalityRange(nextPc, dpc)
14:     // propagate pcf_opt to the plan class pc
15:     pcf_opt ← propagatePcf(pcf_opt, pc.optPlan, nextPc)
16:     // re-enumerate alternative plans in the plan class
17:     declare List⟨Plan⟩ alternatives ← pc.reEnumeratePlans()
18:     // iterate over the alternative plans in the plan class
19:     for each plan in alternatives do
20:         if !isPlanReferencingDpc(plan, dpc) then
21:             declare PCF pcf_stat ← (plan.costs)
22:             ⟨f_I^↓, f_I^↑⟩ ← narrowRange(pcf_opt, pcf_stat, f_I^↓, f_I^↑)
23:             continue
24:         end if
25:         declare PlanClass nextPc
26:         nextPc ← getSubPlanToDpc(plan, dpc)
27:         // get all optimal plans of the sub-plan class
28:         declare OPC tmpOpc
29:         tmpOpc ← findAllOptPlans(nextPc, dpc, f_I^↓, f_I^↑)
30:         for each pcf_alt in tmpOpc do
31:             // propagate pcf_alt to the plan class pc
32:             pcf_alt ← propagatePcf(pcf_alt, plan, nextPc)
33:             ⟨f_I^↓, f_I^↑⟩ ← narrowRange(pcf_opt, pcf_alt, f_I^↓, f_I^↑)
34:         end for
35:     end for
36:     declare OPC cachingOpc(f_I^↓, f_I^↑)
37:     insertIntoOpc(cachingOpc, pcf_opt, true)
38:     pc.cacheOpc(cachingOpc)
39:     return ⟨f_I^↓, f_I^↑, pcf_opt⟩
40: end function
```

next step we re-enumerate all alternative plans for the plan class pc (Line 17), and iterate over each alternative plan (Line 19). In case a plan does not reference the DPC (Line 20), we create a constant cost

Algorithm 3.5 Find All Optimal Plans

```
1: function findAllOptPlans(PlanClass pc, DPC dpc, Cardinality f_I^↓, Cardinality f_I^↑)
2:     // check if there is a cached OPC for this plan class
3:     declare OPC cachedOpc ← pc.getCachedOpc()
4:     if !cachedOpc.empty() then
5:         cachedOpc.updateRanges(f_I^↓, f_I^↑)
6:         return cachedOpc
7:     end if
8:     // create an OPC to return
9:     declare OPC opc(f_I^↓, f_I^↑)
10:    // recursion base
11:    if pc = dpc then
12:        insertIntoOpc(opc, PCF(f_I) = f_I + c_const, true)
13:        return opc
14:    end if
15:    // re-enumerate alternative plans in the plan class
16:    declare List⟨Plan⟩ alternatives ← pc.reEnumeratePlans()
17:    // iterate over all plans in the plan class
18:    for each plan in alternatives ∪ pc.optPlan do
19:        if !isPlanReferencingDpc(plan, dpc) then
20:            declare PCF pcf_stat ← (plan.costs)
21:            insertIntoOpc(opc, pcf_stat, true)
22:            continue
23:        end if
24:        declare PlanClass nextPc
25:        nextPc ← getSubPlanToDpc(plan, dpc)
26:        declare OPC tmpOpc
27:        tmpOpc ← findAllOptPlans(nextPc, dpc, f_I^↓, f_I^↑)
28:        for each pcf_alt in tmpOpc do
29:            pcf_alt ← propagatePcf(pcf_alt, plan, nextPc)
30:            insertIntoOpc(opc, pcf_alt, true)
31:        end for
32:    end for
33:    pc.cacheOpc(opc)
34:    return opc
35: end function
```

PCF (Line 21), and invoke Algorithm 3.1 to potentially narrow the current optimality range (Line 22). In case a plan directly or indirectly references the DPC, we determine the sub-plan, i.e., the plan class that contains the DPC, and invoke Algorithm 3.5 on this plan class (Line 29). Algorithm 3.5, which we further explain below, returns an OPC with PCFs of relevant plans for the considered plan class to Algorithm 3.4. Next, we iterate over each PCF in the OPC (Line 30), propagate it to the current plan class (Line 32), and use Algorithm 3.1

to compare it with the PCF of the optimal plan to further narrow the optimality range (Line 33).

While Algorithm 3.4 is only invoked for plan classes that are part of the optimal plan, Algorithm 3.5 can be invoked for any plan class that directly or indirectly references the DPC, such as RSTU in the example in Figure 3.4. Algorithm 3.5 takes the plan class to consider, the DPC, and the current lower and upper bound as parameters (Line 1). It returns an OPC, i.e., a set of PCFs that are optimal within the given range. Since Algorithm 3.5 can be invoked multiple times for a plan class during one optimality range calculation, it is useful to cache the OPC. So initially, we check if there is an OPC cached for this plan class that can be returned (Lines 3 to 7). Like Algorithm 3.4, Algorithm 3.5 also re-enumerates all plan alternatives for the considered plan class (Line 16), and loops over each alternative plan (Line 18). Plans which do not reference the DPC are immediately inserted as constant cost PCF into the OPC (Lines 19 to 23). The insertion into an OPC was described in Algorithm 3.2. For plans which directly or indirectly reference the DPC, we determine the sub-plan, i.e., the plan class that contains the DPC (Line 25), and recursively invoke Algorithm 3.5 again (Line 27). This recursion ends when the DPC is reached (Lines 11 to 14). When the recursive invocation for a plan class returns an OPC (Line 27), we loop over each plan in the OPC (Line 28), propagate it to the current plan class (Line 29), and insert it into the OPC of the current plan class (Line 30). The current OPC is cached in the plan class (Line 30) and returned (Line 34).

Algorithm 3.5 can be invoked on plan classes that were previously considered by Algorithm 3.4, i.e., plan classes on the path between the final plan class and the DPC. For that reason, Algorithm 3.4 caches OPCs (Lines 36 to 38). In contrast to Algorithm 3.5, the OPC cached by Algorithm 3.4 contains only $\mathtt{pcf}_{\mathtt{opt}}$, because Algorithm 3.4 narrows the current optimality range. Since ranges never become wider, no PCFs except $\mathtt{pcf}_{\mathtt{opt}}$ can be optimal in the current range.

3.2.5 *Complexity Analysis*

We now derive theoretical worst case bounds for the number of enumerated plans required to compute the precise optimality range. We consider bushy plans without cross products. Bushy plans are most complex, and cover all other plan trees. Dynamic programming optimizers for instance can find bushy plans. Linear plans are simpler and covered by bushy plans. The number of enumerated plans first of all depends on the query graph topology: We consider chain queries and star queries. Chain queries are the most simple query graph topology. Star queries are one instance of complex query graph topologies that occur in practice, have far more plan alternatives than chain queries, and are motivated by data warehouses. The number of enumerated

plans for the optimality range calculation also depends on the chosen DPC: The deeper the DPC, the more plans are enumerated. We consider the worst case for a DPC, which is always a base table plan class. The number of enumerated plans is significantly reduced by the OPC pruning strategy, presented in Section 3.2.2. Since we analyze the worst case, we do not consider pruning. In our experiments in Section 3.3.2, we show that pruning significantly reduces the number of enumerated plans.

3.2.5.1 *Chain Queries*

The number of bushy trees without cross products enumerated by dynamic programming optimization for a chain query with n input relations is known to be [14]:

$$(n^3 - n)/6 \tag{3.21}$$

For optimality range calculation, we consider only plan classes that directly or indirectly reference the DPC. But in contrast to dynamic programming, we have to enumerate all possible plans for a plan class, not only those that consist of optimal sub-plans. Pruning may reduce this number, as discussed above. The number of all bushy trees without cross products for a chain query with $n > 1$ relations is known to be [74]:

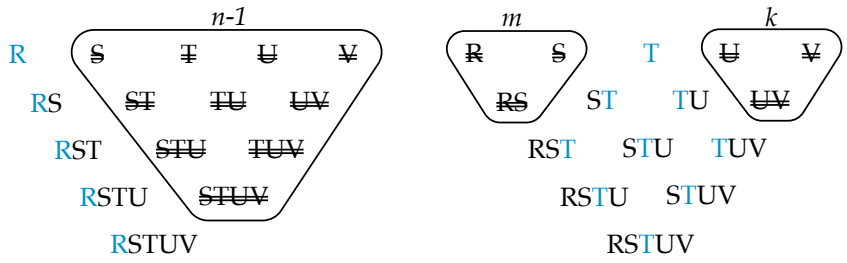
$$2^{n-1}\mathcal{C}(n-1) \tag{3.22}$$

where $\mathcal{C}(n)$ are the catalan numbers. The worst case for the number of enumerated plans is always the DPC being a base table plan class. For chain queries we distinguish two cases: (1) The DPC is the plan class of a base table with only one neighbor in the chain query, as shown in Figure 3.5a. (2) The DPC is the plan class of a base table with two neighbors in the chain query, as shown in Figure 3.5b. Figure 3.5a indicates how to derive the worst case bound for the first case. We subtract the number of all possible plans for $n-1$ relations from the number of all possible plans for n relations. This gives us the worst case bound for the number of bushy plans without cross products enumerated by optimality range calculation for the first case:

$$2^{n-1}\mathcal{C}(n-1) - 2^{n-2}\mathcal{C}(n-2) \tag{3.23}$$

The second case is the actual worst case for chain queries. Figure 3.5b illustrates how we derive the worst case bound. We subtract the number of all possible plans for m and for k relations from the number of all possible plans for n relations, where m and k are:

$$m = \left\lfloor \frac{n-1}{2} \right\rfloor ; k = n - 1 - m \tag{3.24}$$



(a) Special Case (b) General Case

Figure 3.5: Plan class diagrams indicating the theoretical worst case number of enumerated plans for chain queries during optimality range calculation.

Note that m and k are different numbers for even values of n. This results in the following worst case bound for the number of bushy plans without cross products, enumerated by optimality range calculation for chain queries:

$$2^{n-1}\mathcal{C}(n-1) - 2^{m-1}\mathcal{C}(m-1) * 2^{k-1}\mathcal{C}(k-1) \tag{3.25}$$

3.2.5.2 *Star Queries*

In star queries, every dimension table has to be joined with the fact table, i.e., the center of the star. As a result, each plan class beyond the base table level directly or indirectly references the fact table plan class. So the worst case for the number of enumerated plans for star queries, is that the DPC is the fact table plan class. In this case, all plan classes except the other base table plan classes have to be enumerated. This gives us a theoretical worst case bound for enumerated bushy (zig-zag) trees without cross products for star queries, which is identical to the number of all possible plans [74]:

$$2^{n-1}(n-1)! \tag{3.26}$$

In Section 3.3.2, we experimentally show that the actual number of enumerated bushy plans without cross products for chain and star queries is significantly smaller than the worst case bounds.

3.2.6 *Relaxing the Assumptions*

At the beginning of Section 3.2, we made four assumptions to simplify the calculation of optimality ranges: (1) We considered query execution plans containing only scan and join operators. (2) We assumed that the dynamic programming optimizer only stores the cheapest plan per plan class and no additional, more expensive plans with different physical properties. (3) We used a cost function, which creates

PCFs that are linear functions. (4) We assumed the absence of estimation errors in parallel sub-plans. In this section, we discuss the effects when these assumptions do not hold.

3.2.6.1 *Operators other than Joins and Scans*

Optimality range calculation is of course not limited to query execution plans with join and scan operators. The cost of a plan can be modeled as a function of any intermediate result cardinality. We only have to ensure that only plans with the same logical properties are compared, when narrowing the optimality ranges. As additional operators affect the complexity of enumeration algorithms [43, 68, 83, 90], they also affect the complexity of optimality range calculation.

3.2.6.2 *Multiple Optimal Plans per Plan Class*

At the end of the query optimization, there is only one optimal plan for which we calculate the optimality ranges, the cheapest plan in the final plan class. This optimal plan can consist of sub-plans that have properties. So when Algorithm 3.4 reaches a plan class where a plan with properties is selected, it narrows the optimality range (Algorithm 3.1) only with plans that have the same property. Consequently, when Algorithm 3.5 is invoked in such a context, it returns only plans that have at least the corresponding properties: a plan B with at least the same properties and lower cost dominates another plan A, and thus A can be pruned. An example is a plan class RST that has a plan where the ID column is sorted, and a plan class RSTU where a sort merge join on the ID column is optimal. Within the plan class RST we would only narrow the range with plans that have the same sorting on column ID, or any additional property. On the plan class RSTU in contrast, we consider all possible plans, especially all options for the plan class RST. One alternative for the plan class RSTU could be a hash join of the plan classes RST and U. In this case Algorithm 3.5 returns also plans without properties for the plan class RST.

3.2.6.3 *Non-Linear PCFs*

Whether PCFs are linear functions or not, only affects the calculation of plan cost intersections with the narrow range algorithm (Algorithm 3.1), and the OPC design (see Algorithm 3.2). Calculating the intersection point for two non-linear PCFs can be done numerically, for example using the Newton method [39], when we assume monotonically increasing cost functions. Furthermore, narrowing the current optimality range (Algorithm 3.1) could adjust both bounds of the optimality range when there is more than one intersection point within the current optimality range. The objective for OPCs stays the same for non-linear PCFs: contain only PCFs that are piecewise optimal in the current optimality range. During the insertion, it still has

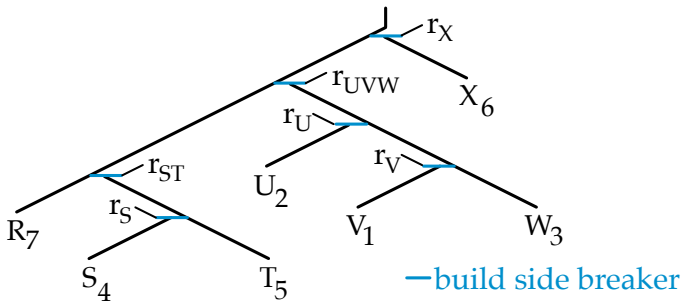


Figure 3.6: Example plan with parallel subtrees.

to be checked if there is at least one point where the new PCF is optimal, and which PCFs in the container became sub-optimal because of the new PCF. This can be also done numerically. The plan space considerations in Section 3.2.3 stay unaffected except that for other cost functions that lead to non-linear PCFs, there might be more plan alternatives per plan class. The optimality range calculation, as described in Section 3.2.4 and the Algorithms 3.3 to 3.5, is unaffected.

3.2.6.4 *Estimations Errors in Parallel sub-plans*

The objective of optimality ranges is to indicate when a query execution plan becomes sub-optimal. This can have multiple reasons, such as transactions modifying the database, or estimation errors. Estimation errors can also impact the correctness of the optimality ranges, since optimality ranges are usually calculated based on estimations. Let us assume a bushy query execution plan with two parallel sub trees, where one sub tree was executed. The executed sub-plan can have a cardinality other than the estimated one, e.g. it could be much smaller, but still in the range. This has an impact on the correctness of the other sub-plan's optimality range, which was calculated based on the statistics available before the execution. One option is to model the cardinalities of parallel sub-plans as parameter in the optimality range, so that the bounds are not just a value but a function. We use Figure 3.6 to explain our solution.

Figure 3.6 shows a query execution plan that joins seven relations. It is a pipelined plan using hash joins. The pipeline breakers are highlighted. The relation subscripts represent the order in which the pipelines are executed, starting with relation V. We calculate the optimality ranges r for the intermediate result of each breaker. In the calculation of the optimality range r_V, the PCFs have just one parameter, the cardinality f_V. At runtime it can happen that f_V is in the range r_V but different than the estimated cardinality $\hat{f}_V$. This has an impact on the correctness of the next range r_U. To consider the potential runtime error of f_V, we leave it as additional parameter in the calculation of r_U. As a result, the PCFs have two instead of one parameter (f_V and f_U), because there is one executed parallel

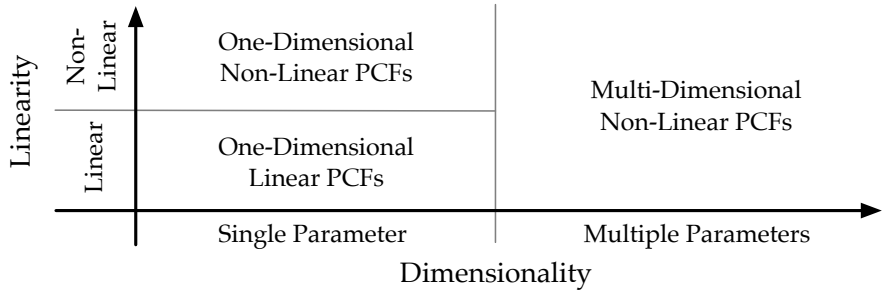


Figure 3.7: Solution space for PCFs in optimality range calculation.

sub tree. For the next range r_{UVW} the PCFs have only one parameter (f_{UVW}), because there are no executed parallel trees at this point. For r_S the PCFs have two parameters (f_S and f_{UVW}), as well as for r_{ST} (f_{ST} and f_{UVW}). In the range calculation for r_X, the PCFs have three parameters (f_X, f_{ST} and f_{UVW}). For the optimality range calculation, this requires the PCFs to support more than one parameter, and different implementations for plan cost intersection calculation and OPCs. At execution time, the potential open parameters in optimality ranges can be set with runtime cardinalities to get the precise lower and upper bound cardinalities. There are also options to avoid the multi-dimensional optimality range calculation. They depend on the application in which the optimality ranges are used: One option is to defer the optimality range calculation to runtime, and use the latest statistics. Some applications, such as plan caching, just need a binary decision if the plan is sub-optimal. Plan caching does not necessarily need to know the edge where the plan becomes sub optimal to evict it from the cache.

Figure 3.7 sums up the solution space for PCFs in optimality range calculation. PCFs can have one or multiple parameters (x-axis), and can be linear or non-linear (y-axis). In this work, we presented cost intersection and the OPC implementation for the lower, left quadrant. The basic principles of the optimality range calculation algorithm, i.e., the bottom up comparison of plan alternatives, can be applied for the remaining quadrants.

3.3 EXPERIMENTAL EVALUATION

We implemented the optimality range calculation in our dynamic programming join optimizer (see Section 2.7). Our join optimizer uses DPSize [4] as dynamic programming enumeration algorithm, C_{out} [26] as cost function, and the basic cardinality estimator (see Section 2.6.3). We use this optimizer to find the initial optimal plan and calculate the optimality ranges. To verify our experimental setup, we show the performance of C_{out} and the cost estimator in Section 2.7. The majority of numbers we report only depend on the considered

algorithm and not on the machine the experiments run on. Reported execution times were taken on a two socket Intel Xeon E5-2660 v3 system with 128 GB of main memory, running a Linux with 3.12.74 kernel. Our dynamic programming optimizer and optimality range calculation implementation are single threaded and compiled with `gcc` version 6.3.0 and optimization option -O3.

3.3.1 *TPC-H Ranges*

In this section, we present optimality ranges for a benchmark similar to TPC-H [89], to show the drawbacks of simple heuristics for optimality ranges, which define an optimality range by just multiplying a constant factor to the estimated cardinality [31, 41]. We further discuss the related work on optimality ranges in Section 3.4. We picked the original schema of TPC-H with scale factor 10, and modified the TPC-H queries to be pure join queries. Note that the optimality range calculation algorithm is not limited to pure join queries. For each query, we searched the optimal query execution plan with the dynamic programming join optimizer. After finding an optimal plan, we kept the dynamic programming table, and calculated the optimality range for each edge in the optimal plan. We present the optimality ranges for the TPC-H-like Queries 8 and 9 in Figure 3.8. We got similar results for all queries, but picked those two queries since they have many joins, and cover all typical cases of optimality ranges we found. The x-axis represents edges of the optimal query execution plan for which we calculated the optimality ranges. The y-axis represents the cardinality in logarithmic scale. We show the lower and upper bound of the optimality range as a box plot, with the estimated cardinality for the considered edge in the middle. In the implementation, the maximum upper bound is limited to $2^{44} - 1$ (instead of ∞).

Multiple aspects are evident: First of all, the width of optimality ranges can vary from very narrow, e.g. Figure 3.8a edge 1, to very wide, e.g. Figure 3.8b edge 5. Additionally, the width of an optimality range is independent of the estimated cardinality, i.e., there are very narrow ranges for small cardinalities, e.g. Figure 3.8a edge 1, but also very narrow ranges for large cardinalities, e.g. Figure 3.8a edge 9. So the assumption that the optimality range grows with estimated cardinality value does not hold. More interesting is the position of the estimated cardinality in the optimality range: There are estimated cardinalities, which are close to the lower bound of the optimality range, e.g. Figure 3.8a edge 6, or close to the upper bound, e.g. Figure 3.8b edge 4. Consequently, the optimality range bounds cannot be defined by multiplying a constant factor to the estimated cardinality. Furthermore, the cases in which the estimated cardinality is close to a lower or upper bound show that there are situations in query processing, where even small estimation errors lead to a different optimal plan.

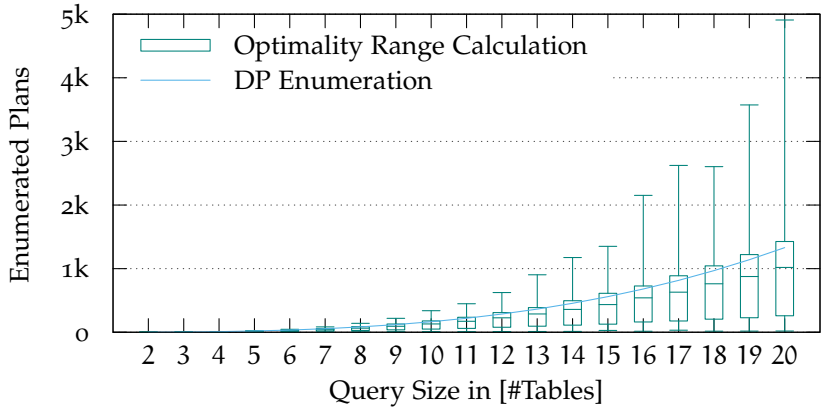
(a) Query 8

(b) Query 9

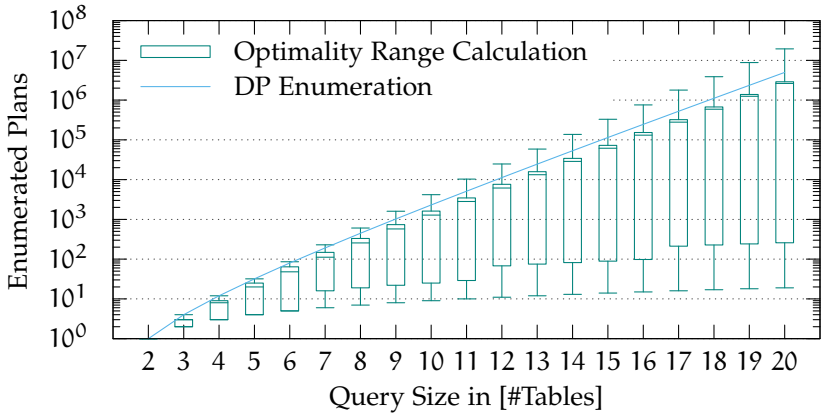
Figure 3.8: Optimality ranges and estimated cardinalities for the optimal plans of two TPC-H-like queries.

3.3.2 *Enumerated Plans*

To experimentally evaluate the number of enumerated plans, we randomly created chain and star queries. We picked chain queries, because they are the most simple query grapgh topology. Star queries are one instance of complex query graph topologies that occur in practice, have considerably more plan alternatives than chain queries, and are motivated by data warehouse queries. For both topologies, we experimented with query sizes, i.e., numbers of joined relations, from 2 to 20. For each topology and query size, we created 100 random databases. The base relation cardinalities are random numbers between 10^4 and 10^8. The join cardinalities between two relations are random numbers between $max(|R_i|, |R_j|) - 1000$ and $max(|R_i|, |R_j|) + 10$ for two relations R_i and R_j. For each randomly generated database, we searched the optimal plan for the corresponding query and calculated the optimality ranges on all edges of the optimal plan.



(a) Chain Query

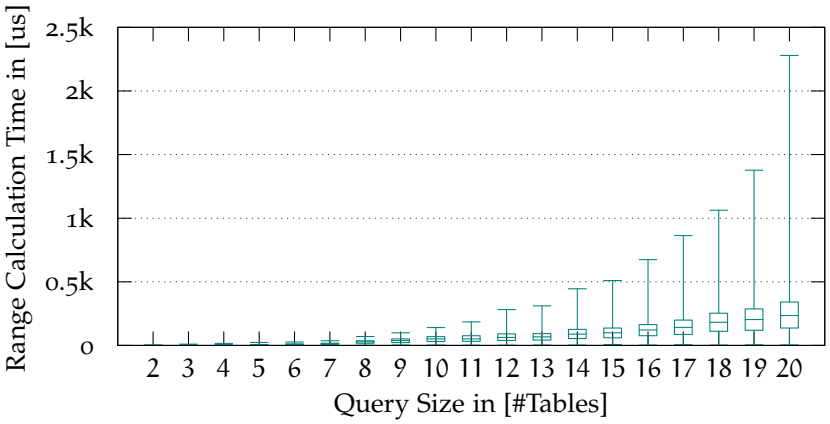


(b) Star Query

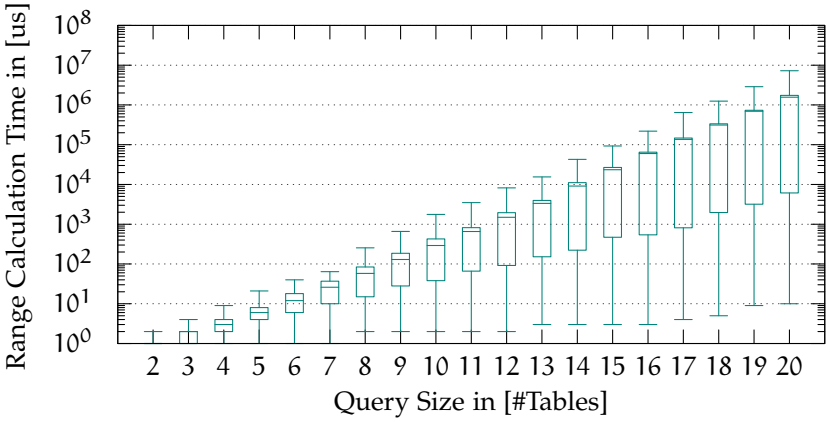
Figure 3.9: Enumerated plans during optimality range calculation for each edge of 100 random queries.

We show the number of enumerated plans during the optimality range calculation in Figure 3.9. Figure 3.9a shows the results for chain queries, and Figure 3.9b for star queries. The x-axis is the query size in number of relations, and the y-axis the number of enumerated plans. The y-axis in Figure 3.9b has a logarithmic scale. We show the number of enumerated plans for calculating the optimality ranges as box plots. Each entry in a box plot is a single optimality range calculation on one edge in one of the 100 optimal query execution plans. The percentiles of the box plots are 0%, 25%, 50%, 75%, and 100%. As a baseline we show the number of enumerated plans by dynamic programming query optimization. This number is the same for queries with equal topology and size.

Figure 3.9 shows that the query size has the highest impact on the number of enumerated plans during optimality range calculation. The remaining spread is caused by the following three factors: (1) the depth of the DPC, (2) the structure of the query execution plan, e.g. a linear tree, a balanced bushy tree, or something in between, (3) the



(a) Chain Query



(b) Star Query

Figure 3.10: Optimality range calculation time for each edge of 100 random queries.

pruning strategy. All three factors interact, and especially the pruning makes it hard to do a decoupled analysis. We see a correlation between the number of enumerated plans and the depth of the DPC. Usually the deeper the DPC, the larger the number of enumerated plans. The 100% percentiles in Figure 3.9 are usually optimality range calculations where the DPC is one of the deepest edges in the query execution plans. Nevertheless, there are cases in which the optimality range converges after a few considered plan alternatives. This enables an effective pruning in the low level plan classes, which is beneficial for the eventually enumerated number of plans. In these cases it could happen that the worst case number of enumerated plans is not for calculating the optimality range on one of the deepest edges in the query execution plan. Also, the structure of the query execution plans has an impact on the number of enumerated plans because it affects the maximum depth of an edge or DPC. The structure has mainly an impact on chain queries, since they can result in anything from a linear, e.g. left deep query execution plan to a balanced bushy tree. For star

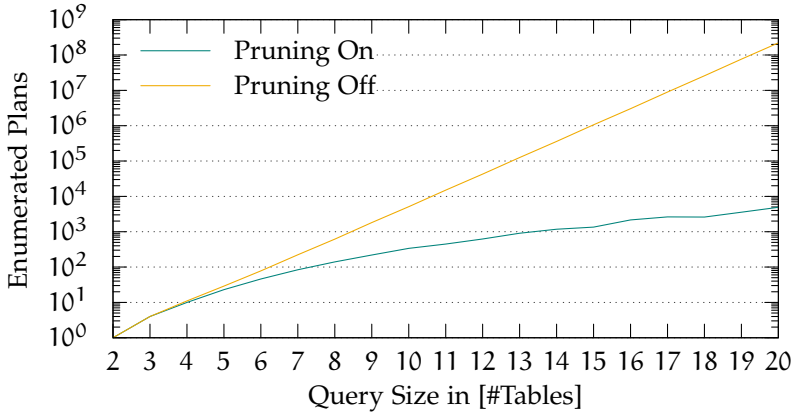
queries, the structure has no impact, since star queries always result in linear plans when cross products are ignored.

The experimental results in Figure 3.9a and 3.9a also show that the median number of enumerated query execution plans for up to 20 joined relations is smaller than the number of plans enumerated by dynamic programming optimization. The experimental worst case can be larger than dynamic programming optimization, but is still significantly smaller than the theoretical worst case boundaries derived in Section 3.2.5. For chain queries, we enumerate at most four times more query execution plans than dynamic programming optimization. Note, that dynamic programming optimization, and hence also the calculation of optimality ranges, enumerate only a polynomial number of query execution plan alternatives with respect to the number of input relations. This is significantly smaller than the theoretical worst case bound, which grows exponentially.

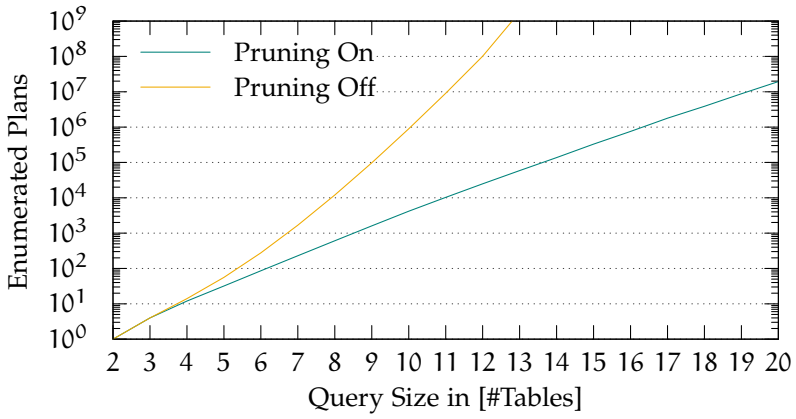
In addition to the number of enumerated plans, we show the optimality range calculation times in Figure 3.10. The setup is the same as in Figure 3.9 except that the y-axis shows the calculation time in microseconds. The calculation times in Figure 3.10 show a strong correlation to the number of enumerated plans in Figure 3.9. Calculating an optimality range took at most 2.3 milliseconds for a chain query, and 7.3 seconds for a star query with 20 relations.

The positive enumeration behavior of optimality range calculation is due to its pruning strategy. Without pruning, all possible plan alternatives for a considered plan class would be enumerated. We further investigate the impact of pruning in the next experiments: We choose the same setup as for the experiments in Figure 3.9, i.e., 100 random databases for chain and star queries and different query sizes, but compare the number of enumerated plans with enabled and disabled pruning. We present the worst case number of enumerated plans in Figure 3.11a for chain queries and Figure 3.11b for star queries. Note that the results for enabled pruning correspond to the 100% percentiles in Figure 3.9. The y-axis in both Figure 3.11a and 3.11b is logarithmic, since the number of enumerated plans for chain queries with disabled pruning is not polynomial anymore, but exponential. We also set an execution time limit of two minutes. This means the results for a random query were ignored whenever calculating the optimality range for one of its edges took longer. This was the case in all star query experiments without pruning and more than 13 tables.

The results in Figure 3.11 show the relevance of pruning. For chain queries the number of enumerated plans without pruning is at most 220,565,946, while it is at most 4,908 with pruning. For star queries, we consider query size 13, where 1,228,418,488 plans are enumerated without pruning and 58,733 with pruning. The results show the impact of our pruning strategy, which is one contribution of this work.



(a) Chain Query



(b) Star Query

Figure 3.11: Worst case number of enumerated plans during optimality range calculation with and without pruning.

3.4 RELATED WORK

Multiple topics in query processing are related to optimality ranges. Therefore, we categorize related work on optimality ranges into the following three fields: *Adaptive Query Processing*, *Parametric Query Optimization*, and *Offline Plan Space Analysis*.

3.4.1 *Adaptive Query Processing*

The question if a given query execution plan remains optimal in the presence of cardinality estimation errors, is at the core of *Mid-Query Re-Optimization* [31, 39, 41]. Mid-Query Re-Optimization utilizes statistics that are taken at query execution time to compensate sub-optimal query execution plans caused by cardinality estimation errors. One common approach is to define bounds for intermediate result cardinalities. If the true cardinality at query execution time is out of its bounds, the query optimizer is invoked again to search

for a better plan. Markl et al. [39] called their bounds *validity ranges*. *Dynamic Re-Optimization* by Kabra and DeWitt [31] and *Proactive Re-Optimization* by Babu et al. [41] just define heuristically chosen optimality ranges: They assign an uncertainty value U to an estimate E, and define the lower bound $f_I^{\downarrow} = E - (E * 0.1 * U)$ and the upper bound $f_I^{\uparrow} = E + (E * 0.2 * U)$. Kabra and Dewitt [31] and Babu et al. [41] furthermore present a set of rules to derive if the uncertainty value U of an estimate is low, medium or high, but do not specify actual values for U. Section 3.3.1 shows the limitations of this approach. *Progressive Optimization* by Markl et al. [39] improves on Dynamic Re-Optimization and Proactive Re-Optimization by taking alternative plans into account. To calculate a validity range bound, Progressive Optimization models the costs of the optimal plan and the considered alternative plan as a function of one cost parameter, i.e., as PCF. In the next step, Progressive Optimization calculates the intersection point of the two PCFs using a numeric approach, i.e., a modified Newton-Raphson. This gives potentially a new validity range bound (see Section 3.2.1). Progressive Optimization compares only structurally equivalent plans, i.e., plans where only the operator implementation and input order are different, and ignores plans with alternative operator orders. In contrast to Progressive Optimization, our work takes all necessary plan alternatives into account. Consequently, our optimality ranges would trigger a re-optimization if and only if a better plan alternative exists.

3.4.2 *Parametric Query Optimization*

Such as Progressive Optimization, our approach has similarities to *Parametric Query Optimization* (PQO) by Hulgeri and Sudarshan [37, 38], which determines a set of query execution plans that are optimal for different values of one or multiple cost parameters. PQO also models the costs of plans as PCF, and selects the optimal plans for the ranges in the parameter space. The weak spot of PQO is the combinatorial explosion of alternative plans. Compared to PQO, the calculation of optimality ranges is only interested in the bounds of the optimal plan. In order to find the exact optimality range, our work in theory has to enumerate the same number of plans as PQO. But since we are only interested in the optimality range, and not the plans that are optimal beyond the range, we can effectively prune. Section 3.3.2 shows the effectiveness of our pruning strategy, and proves that it makes our approach applicable in practice. *Progressive Parametric Query Optimization* (PPQO) by Bizarro et al. [55] is a more practical PQO approach, which caches the optimal plans for already executed parameter combination. To decide whether a parametric query with given parameters can be calculated by one of the cached plans, PPQO

heuristically assign selectivity bounds to each cached query execution plan that are similar to optimality ranges.

3.4.3 *Offline Plan Space Analysis*

Plan Diagrams by Haritsa et al. [45, 53] are a pictorial enumeration of alternative query execution plans. A plan diagram indicates the selectivities in which different query execution plans are optimal. Each of the usually two dimensions in a plan diagram represents a predicate selectivity, which is varied from zero to one. For each of the resulting selectivity points, the optimal plan is searched. Equal execution plans for different selectivity points in the diagram are illustrated in the same color, so that clusters of plans become visible. In the end, each plan cluster indicates in which selectivity area a plan is optimal. While Plan Diagrams consider all plans that are optimal within the selectivity space, our work is focused on only one optimal plan at-a-time. But our approach is not limited to two selectivities. We can determine the optimality ranges for all operator selectivities in the considered optimal plan. Furthermore, our approach is more analytic using plan space and cost parameter considerations. Plan Diagrams in contrast cannot be used at runtime of the query, since it is a brute force approach, which searches the optimal plans for all selectivity combinations. It is a tool to analyze properties of a query optimizer in offline mode. Offline approaches similar to Plan Diagrams are *Plan Bouquets* by Dutt and Haritsa [71] and *Exact Cardinality Estimation* by Chaudhuri et al. [56].

3.5 APPLICATIONS IN QUERY PROCESSING

We consider our optimality range approach as an important building block for query processing, which helps to improve applications such as execution Plan Caching, Parametric Query Optimization, and Mid-Query Re-Optimization, described below.

3.5.1 *Execution Plan Caching*

While a traditional data warehouse system may be able to cache plans for all executions of a statement between data loading processes, this strategy turns out to be unsuitable for HTAP. On the one hand, it is improbable that a single row update, as they are typical for transactional workloads, makes a plan sub-optimal. On the other hand, it is almost certain that the constant change in the database makes a plan at some point sub-optimal. This illustrates a new challenge that arises especially in HTAP systems: Is a cached plan still optimal or not? Optimality ranges can solve this problem, since they can be used as caching strategy for reoccurring queries. We calculate the optimal-

ity ranges for an optimal plan, and add the plan with ranges to the query cache. Each time a cached query is executed, the true cardinalities for the edges, i.e., the intermediate results are available. Once a cardinality is out of its optimality range, the cached plan is not optimal anymore and evicted from the cache. It is also promising to use Robust Query Processing techniques [80] such as LEO [36] to find a good plan that is annotated and cached.

3.5.2 *Parametric Queries*

Progressive Parametric Query Optimization [55] describes an solution for Parametric Query Optimization that avoids searching optimal plans for different parameter combinations at optimization time. It caches optimal plans for already executed parameter combination, called cost point. Cache hits can be either exact matches for a cost point or in a heuristically defined distance to a cost point in the cache. This approach can be improved with optimality ranges by storing a range in which a plan is optimal instead of a cost point. It reduces the number of stored plans, and enables a more precise plan choice.

3.5.3 *Plan Robustness*

Although it is not the primary objective of optimality ranges, they can be used to evaluate the robustness of optimal query execution plans. Criteria are the width of optimality ranges and the positions of the estimated cardinalities in the ranges. A wide optimality range and an estimated cardinality in the middle of the optimality range for every edge in a query execution plan indicate a high robustness towards estimation errors. A narrow optimality range, or an estimated cardinality close to a bound of the optimality range indicate that already small estimation errors lead to a cheaper plan. In this case the robustness towards cardinality estimation errors is low. In Chapter 4 we present a more general approach to evaluate the robustness of query execution plans, which is not limited to optimal query execution plans. Its foundation is the plan cost intersection of optimality range calculation, which we explained in this chapter.

3.5.4 *Mid-Query Re-Optimization*

The calculation of precise optimality ranges is a useful tool to validate of Mid-Query Re-Optimization strategies. Section 3.3.1 demonstrates the shortcomings of the existing Mid-Query Re-Optimization heuristics: They fail to characterize optimality ranges, and consequently miss the switch to cheaper plans. Our Experiments in Chapter 5 show that the query execution time increases due to missing cheaper plans can be significant. Unfortunately, the worst case complexity of an op-

timality range calculation is worse compared to a dynamic programming optimization, and our experiments show that pruning can reduce the number of enumerated plans in optimality range calculation only to the same order of magnitude as dynamic programming optimization (see Section 3.3.2). This makes optimality ranges unsuitable as Mid-Query Re-Optimization criterion for ad-hoc queries. Nevertheless, the insights gained from the experiments on optimality ranges build the foundation for our improved Mid-Query Re-optimization strategy presented in Chapter 5.

3.6 CONCLUSION

In this chapter, we study the formal and conceptual foundations for the calculation of exact optimality ranges for intermediate result cardinalities in optimal query execution plans. Next to the formal algorithm description, we derive worst case boundaries for the number of enumerated plans during optimality range calculation for chain and star queries. Our experimental evaluation shows that the actual worst case number of enumerated plans for queries with up to 20 relations is significantly smaller than the theoretical worst case boundaries. This is due to our pruning strategy. In our experiments, pruning reduces the worst case number of enumerated plans for chain queries with 20 relations from 220,565,946 to 4,908. Our study on optimality ranges for a TPC-H-like benchmark demonstrates the limitations of simple heuristics. The width of optimality ranges is independent of the estimated cardinality, and varies from narrow to wide. In addition, the position of the estimated cardinality in the optimality range can vary. It can be close to the optimality range boundaries, so that even small estimation errors lead to cheaper plans. Compared to existing approaches, our solution does not rely on simple heuristics, and takes all necessary plan alternatives into account to calculate exact optimality ranges. Nevertheless, our experiments on enumerated plans and calculation time have shown that it is applicable in practice.

This makes optimality ranges a valuable and generic building block for query processing. Optimality ranges can be used as caching strategy to decide when a query execution plan has to be evicted. They can also improve existing approaches in parametric query optimization, and determine the robustness of optimal query execution plans with respect to cardinality estimation errors. Last but not least, optimality ranges help to validate adaptive query processing approaches.

4

ROBUSTNESS METRICS AND ROBUST PLAN SELECTION

Next, we address sub-optimal query execution plans that are caused by cardinality estimation errors. Query optimizers commonly select the cheapest query execution plan based on cardinality estimations. If the cardinality estimations are wrong, the selected query execution plan can have an undesirable behavior. In this chapter, we address this issue with a new plan selection strategy, which takes potential cardinality estimation errors into account.

4.1 INTRODUCTION

Although query optimization is a well-studied problem with numerous approaches being proposed and developed since Selinger's seminal work [4], finding a good query execution plan is still a challenge, even for mature commercial systems. Typically, two major problems arise in this context:

1. A significant increase of query execution times, if the chosen query execution plan turns out to be sub-optimal or even bad, as the experiment in Section 1.1 shows.

2. An unpredictable query execution time behavior due to small changes in the database, which can cause the selection of a fundamentally different query execution plan with a very different query execution time.

Both are problems of *robustness*, which has become an important research topic in query processing. It is discussed in multiple Dagstuhl seminars on *Robust Query Processing*, organized by Graefe and colleagues [60, 65, 91]. Although robust query processing has several aspects ranging from query planning to execution and scheduling, both problems share a common issue related to query robustness: errors in cardinality estimation as a central parameter of a cost model.

Despite the issues of cardinality estimation are evident [78, 93], the majority of query optimizers still chooses the estimated cheapest plan based on the cost model as optimal plan. Potential cardinality estimation errors are not taken into account when choosing a plan.

The major contributions of this chapter are three novel metrics for the robustness of relational query execution plans with respect to cardinality estimation errors. They can assign a numeric value for the robustness of a plan, which can be considered next to the estimated

cost in the selection of a plan. Our metrics support all kinds of operators, operator implementations, query execution plan trees, and monotonically increasing and differentiable cost functions. In contrast to other approaches for robustness plan selection, we can evaluate the plan robustness at query optimization time [53, 71, 82], and assign robustness values independent of other query execution plans [59, 81]. We are also not limited to certain tree structures of query execution plans [77, 81], or consider only plans that are optimal for some cardinalities [40, 41, 53, 82].

Next to the robustness metrics, which we present in Chapter 4.4, there are further contributions: We show a formal problem description and consistency requirements for plan robustness metrics in Section 4.2. Another contribution is the new plan selection strategy based on our robustness metrics that we explain in Section 4.5. As final contribution, we share our experimental evaluation for runtime and robustness of our plan selection strategy, using a synthetic and a real-world database benchmark in Section 4.6.

4.2 FORMAL PROBLEM DESCRIPTION

Due to cardinality estimation errors, the estimated optimal plan chosen by conventional query optimizers frequently fails to be the fastest plan. We argue that choosing a robust plan can result in faster query execution times in the presence of cardinality estimation errors. We formalize the problem of finding a query execution plan that is robust with respect to cardinality estimation errors.

Definition 4.1. *The **cost error factor** c_{err} is the absolute quotient of estimated cost $\hat{c}$ and true cost $\mathring{c}$ of a plan.*

$$c_{err} = \begin{cases} \mathring{c}/\hat{c} & \text{if } \mathring{c} \geqslant \hat{c} \\ \hat{c}/\mathring{c} & \text{otherwise} \end{cases} \qquad \text{where} \quad \mathring{c} > 0 \quad \text{and} \quad \hat{c} > 0$$

A small c_{err} can mean that there were no estimation errors in the plan, because the estimated cost is close to the true cost. If we assume that there are estimation errors in a plan, a small c_{err} means that the plan is robust, because the cost of the plan did not change much in the presence of estimation errors. Consequently we define:

Definition 4.2. *The **most robust plan** is the plan with the smallest cost error factor c_{err} within the set of robust plan candidates.*

Since the true cost $\mathring{c}$ is unknown at optimization time, the cost error factor c_{err} cannot be calculated. Therefore, we approximate it.

Definition 4.3. *A **robustness metric** assigns a robustness value to each robust plan candidate. Ideally, the robustness value is an approximation for the upper bound of c_{err}.*

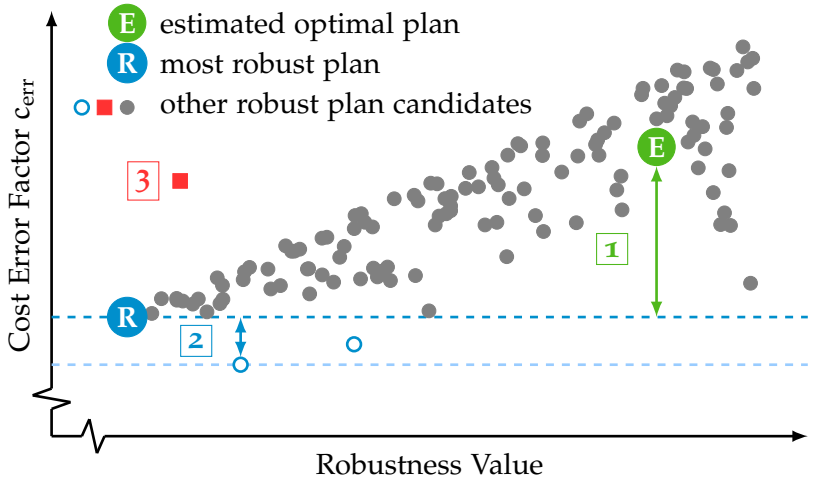


Figure 4.1: Illustrations of consistency requirements for robustness metrics, showing candidate plans with their assigned robustness value and their c_{err}.

Figure 4.1 illustrates the behavior of an ideal robustness metric. The robustness value assigned by the robustness metric is denoted on the x-axis. The y-axis denotes the cost error factor c_{err} (see Definition 4.1). Furthermore, Figure 4.1 shows all robust plan candidates with an assigned robustness value and their cost error factor c_{err}. The estimated optimal plan is highlighted as Ⓔ. The most robust plan according to the robustness metric, i.e., the leftmost plan, is depicted as Ⓡ. We argue that an ideal robustness metric should fulfill the following three consistency requirements.

1 **Cost Error Factor Improvement:** Compared to the estimated optimal plan, the most robust plan according to the robustness metric should always achieve a smaller cost error factor c_{err}.

2 **Cost Error Factor Dominance:** The most robust plan according to the robustness metric dominates all robust plan candidates with respect to the cost error factor c_{err}. This means there should be no plan with a smaller cost error factor c_{err} than the most robust plan, e.g., the empty circle ○ plans in Figure 4.1.

3 **Correlated Cost Error Factor Limit:** A robustness metric should give an upper bound for the cost error factor c_{err} of a plan. Plans with a large robustness value can have a large c_{err}, and plans with a small robustness value should have a small c_{err}. Plans, such as the square ■ plan in Figure 4.1 indicate a suboptimal robustness metric, because the metric classified the plan to be much more robust than it is. The upper bound of c_{err} should be proportional to the robustness value, but c_{err} itself does not have to be proportional to the robustness value, because the estimations can be precise and result in a smaller c_{err}.

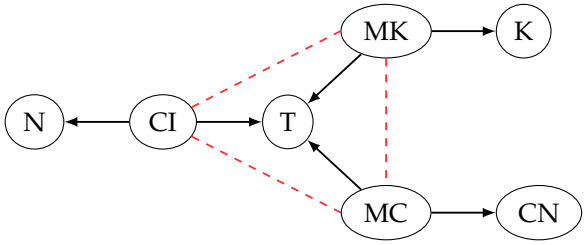


Figure 4.2: Query graph of Join Order Benchmark [78] Query 17, joining the tables `NAME` (N), `CAST_INFO` (CI), `TITLE` (T), `MOVIE_KEYWORD` (MK), `KEYWORD` (K), `MOVIE_COMPANIES` (MC), and `COMPANY_NAME` (CN).

In practice, there is a trade-off between plan robustness and execution time. On the one hand, a robust plan is less sensitive to estimation errors, but not necessarily fast. On the other hand, a fast plan is not necessarily robust. In Section 4.5, we present a robust plan selection strategy that balances plan robustness and execution time.

4.3 ROBUST PLAN EXAMPLE

Figure 4.2 shows the query graph of a real-world query, which we use to illustrate the impact of cardinality estimation errors on the query execution plan selection. We consider Q17 of the Join Order Benchmark (JOB) [78] as a pure join query with a filter on all `movie_id` columns (see Section 4.6). Foreign key joins (1:n) are represented by solid edges with the arrow pointing to the referenced key. Dotted edges denote m:n joins. Of course, we support all kinds of operators and operator implementations. In this example, we use the C_{out} [26] cost function (see Sections 2.6.2 and 2.7). Apart from C_{out}, our robustness metrics support every kind of monotonically increasing and differentiable cost function such as C_{mm} [93], which we use for the experimental evaluation.

In addition to the estimated and true cardinality $\hat{f}$ and $\mathring{f}$, and the estimated and true cost $\hat{c}$ and $\mathring{c}$, we define the estimated selectivity $\hat{s}$, and the true selectivity $\mathring{s}$. We also define the absolute cardinality error Δf, the absolute cost error Δc, and the q-error [58], i.e., the absolute quotient of estimated and true cardinality:

$$\Delta f = \mathring{f} - \hat{f} \qquad \Delta c = \mathring{c} - \hat{c}$$

$$\text{q-error} = \begin{cases} \mathring{f}/\hat{f} & \text{if } \mathring{f} \geqslant \hat{f} \\ \hat{f}/\mathring{f} & \text{otherwise} \end{cases}$$

Figure 4.3 shows the estimated cheapest query execution plan for JOB Query 17, identified by our query optimizer, which we describe in Section 2.7. We argue that our join optimizer's estimated optimal plan choice is very similar to the choice of popular free and commercial systems, due to its enumeration algorithm, the cardinality esti-

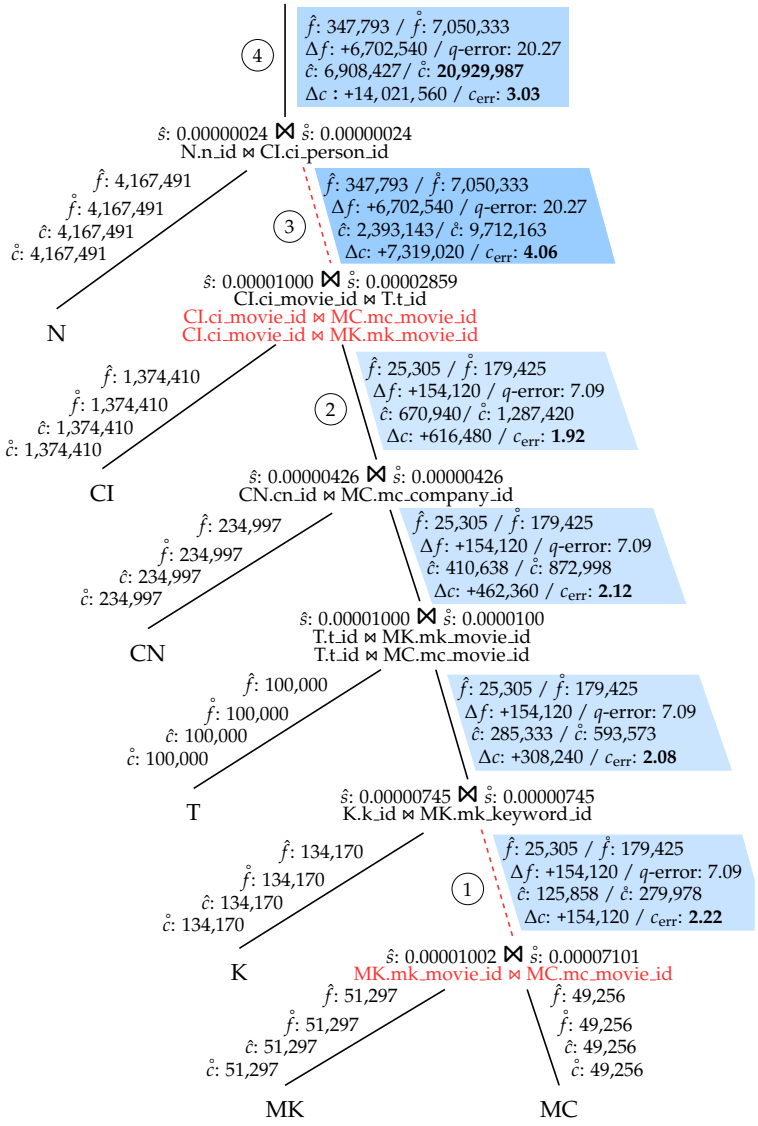


Figure 4.3: Estimated cheapest plan for JOB Query 17, joining the tables NAME (N), CAST_INFO (CI), TITLE (T), MOVIE_KEYWORD (MK), KEYWORD (K), MOVIE_COMPANIES (MC), and COMPANY_NAME (CN).

mation error it creates, and the correlation of its cost functions to the query execution times in our engine (see Section 2.7). Every edge in the query execution plan represents an intermediate result with estimated and true statistics. The true statistics of the final edge 4 shows that the true cardinality of the estimated optimal plan is underestimated by a factor of 20.27 (q-error), and the true costs by a factor of 3.03 (c_{err}). In more detail, we see that the cardinality estimator underestimates two edges in the query execution plan, i.e., the dashed edges 1 and 3. The first join is a m:n join between MOVIE_KEYWORD and MOVIE_COMPANIES. The estimated cardinality on the outgoing edge 1 of this join is 25,305. After executing this plan, it turns out that this

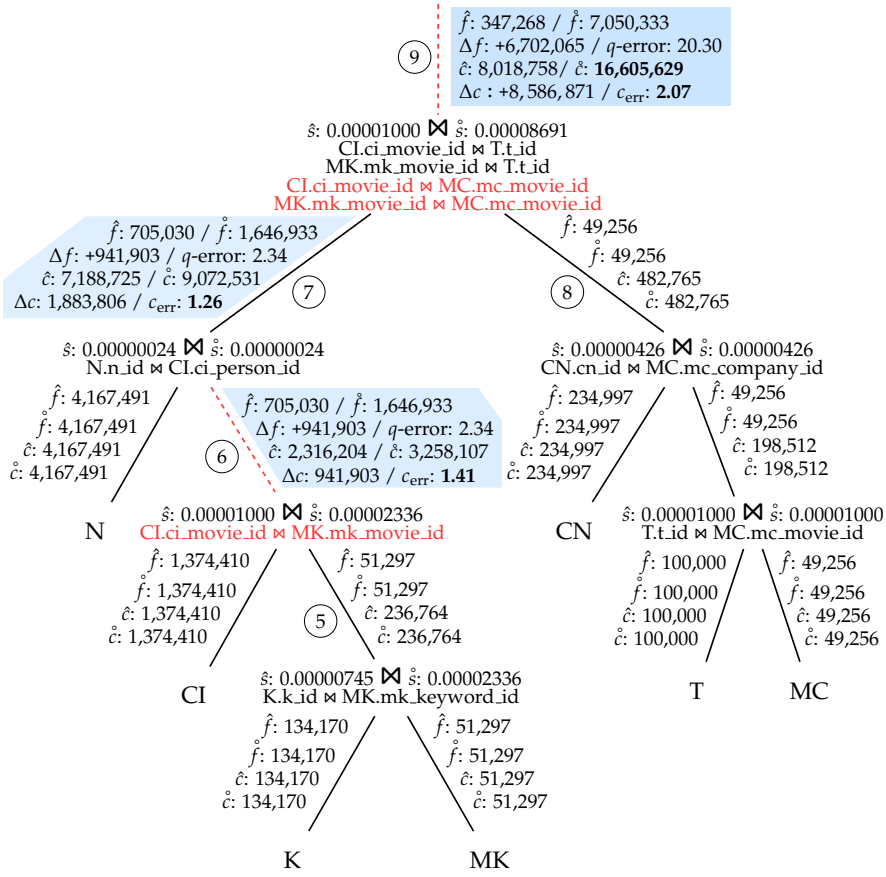


Figure 4.4: Estimated most robust plan for JOB Query 17, joining the tables `NAME` (N), `CAST_INFO` (CI), `TITLE` (T), `MOVIE_KEYWORD` (MK), `KEYWORD` (K), `MOVIE_COMPANIES` (MC), and `COMPANY_NAME` (CN).

is an expanding join, and the cardinality was underestimated, due to missing information about the data distribution. The true cardinality is 179,425, which results in a q-error of 7.09 and in a c_{err} of 2.22. The second underestimation occurs in the join between the `CAST_INFO` table and the subtree of edge 2. Beside a foreign key join, there are two m:n joins involved. Again, this is an expanding join, and the output cardinality is underestimated: The estimated output cardinality is 347,793, but the true cardinality is 7,050,333. Accordingly, the q-error is 20.27 and the c_{err} is 4.06. All other plan edges are estimated correctly, since the q-error is not growing with respect to the child edges. The reason is that all those joins have a foreign key relation, for which the estimation is more precise.

Figure 4.4 shows the estimated most robust plan that our approach selects for Query 17 of the Join Order Benchmark. While the estimated optimal plan in Figure 4.3 has smaller estimated cost, 6,908,427 compared to 8,018,758, the plan chosen by our approach has a lower execution time in the presence of cardinality estimation errors. The major difference between the estimated optimal plan in Figure 4.3

and the estimated most robust plan in Figure 4.4 is the deferred execution of m:n joins. Therefore, the first cardinality estimation error does not occur at the first join, as in the estimated optimal plan on edge 1 in Figure 4.3. In contrast, there are no cardinality estimation errors in the subtrees of the edges 8 and 5 in the estimated most robust plan. Their foreign key joins are estimated correctly. The first underestimation occurs in the join of the `CAST_INFO` table and the subtree of edge 5, i.e. on edge 6. While the estimated cardinality is 705,030, the true cardinality is 1,646,933. Hence, the q-error is 2.34 and the c_{err} is 1.41. The second underestimation occurs at the final join between the subtrees of the edges 7 and 8, in which the last two m:n joins are involved. As a result, the q-error is 20.30 and the c_{err} is 2.07.

Comparing the two plans shows that the c_{err} of the estimated most robust plan, 2.07, is smaller than the c_{err} of the estimated optimal plan, 3.03. Also, the true cost of the estimated most robust plan, 16,605,629, is considerably smaller than the true cost of the estimated optimal plan, 20,929,987. As a result, the estimated most robust plan has a query execution time of 475 ms, which is a speedup of factor two compared to the estimated optimal plan with an execution time of 995 ms. In Section 4.6, we show more complex queries with larger speedups in the presence of cardinality estimation errors.

4.4 ROBUSTNESS METRICS

In this section we answer the question: can we define metrics to quantify the robustness of query execution plans before execution? After execution, the robustness of a query execution plan or a sub-plan can be quantified by the q-error or the c_{err}. In order to quantify the robustness of a plan before execution, deriving Parametric Cost Functions (PCFs) is the first building block. We explain the calculation of PCFs in Section 2.6.4. PCFs are also an essential building block in the calculation of optimality ranges (see Section 3.2.1).

Figure 4.5 shows PCFs modeled as a function of cardinality on a single edge in the plan. There is a PCF for a volatile plan, PCF_{vol}, and for a robust plan, PCF_{rob}. The x-axis denotes the cardinality of the edge and the y-axis the costs of plans. It also shows the estimated cardinality $\hat{f}$ and the true cardinality $\mathring{f}$. Furthermore, it shows the estimated cost $\hat{c}$ and the true cost $\mathring{c}$ for both plans. A robust plan does not necessarily have the smallest cost at $\hat{f}$. In this case the robust plan is not the optimal plan, and the optimal plan not necessarily a robust plan. In the presence of estimation errors, the true cost of a volatile plan can rapidly increase or decrease. In contrast, the true costs for a robust plan are close to the estimated costs in the presence of cardinality estimation errors, i.e., a more moderate slope of PCF_{rob} compared to PCF_{vol}. Therefore, the slopes of PCFs around the estimated cardinality indicate the sensitivity of a plan towards estimation errors. If

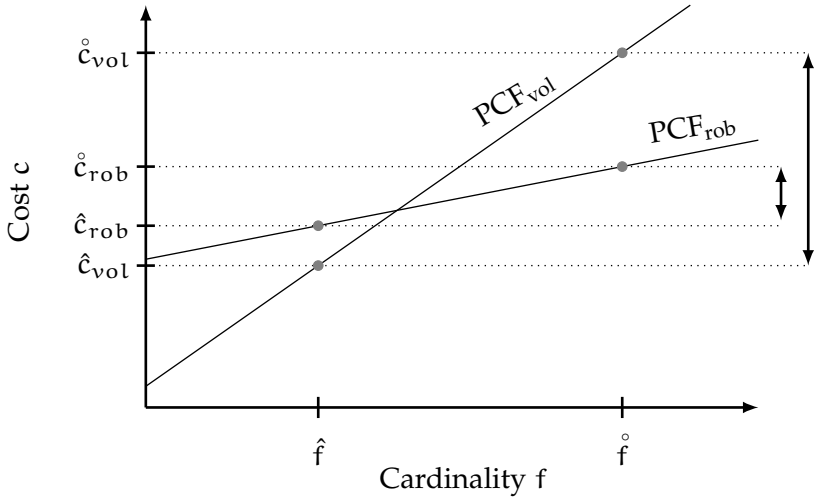


Figure 4.5: Cost behavior of a volatile and a robust query execution plan in the presence of cardinality estimation errors.

the true cardinality $\mathring{f}$ is underestimated, as in Figure 4.5, picking the robust plan will also result in an execution time improvement.

Conventional query optimizers select the plan with the smallest estimated cost, but they do not consider the cost behavior in the presence of estimation errors. Consequently, the estimated optimal plan is not necessarily a robust plan. We argue that considering the cost behavior, i.e., the slopes of a PCF in the plan selection, helps to identify more robust plans. Modeling the C_{out} cost of a plan as a function of one cost parameter for example, results in a linear PCF, i.e., a PCF with the same slope at every cardinality. In contrast, using a cost function other than C_{out} can result in a non-linear PCF. We support non-linear PCFs that are monotonically increasing and differentiable, i.e., have no jumps a slope value at each cardinality.

Next, we discuss examples for PCFs. We consider the estimated optimal plan P_{opt} and the estimated most robust plan P_{rob} for Query 17 of the Join Order Benchmark. Figure 4.3 shows P_{opt}, and Figure 4.4 P_{rob}. For the C_{out} cost function and the given statistics, P_{rob} has estimated costs of $8,018,758$. We assume that the output cardinality of edge 6 in Figure 4.4 is not $705,030$ but an arbitrary value. Let us denote this variable as $f_{CI,K,MK}$ for the output cardinality of joining `CAST_INFO` (CI), `KEYWORD` (K), and `MOVIE_KEYWORD` (MK). Now we model the C_{out} costs of P_{rob} as a PCF on the variable $f_{CI,K,MK}$, i.e., not set $f_{CI,K,MK} = 705,030$ but leave it as parameter when calculating the C_{out} cost of P_{rob}, as explained in Section 2.6.4:

$$C_{out}(P_{rob}, f_{CI,K,MK}) = 2.49 \cdot f_{CI,K,MK} + 6,261,430 \tag{4.1}$$

For each additional output tuple on the $f_{CI,K,MK}$ edge, the total cost of P_{rob} increases by 2.49. The $f_{CI,K,MK}$ edge is the deepest edge in P_{rob}, which contains a $m{:}n$ join. Figure 4.3 shows P_{opt}, in which

edge 1 is the deepest edge with a $m{:}n$ join. The cardinality of edge 1 is denoted as $f_{MK,MC}$. To determine the sensitivity of P_{opt} towards cardinality estimation errors for $f_{MK,MC}$, we calculate the costs of P_{opt} as a PCF on the variable $f_{MK,MC}$:

$$C_{out}(P_{opt}, f_{MC,MK}) = 31.49 \cdot f_{MC,MK} + 6,111,621 \tag{4.2}$$

Equation 4.2 shows that one additional tuple for $f_{MK,MC}$ increases the total cost of P_{opt} by 31.49. Therefore, the edge $f_{MK,MC}$ in P_{opt} has a steeper slope than the edge $f_{CI,K,MK}$ in P_{rob}. Consequently, we consider P_{rob} as a more robust plan, because cardinality deviations have smaller impact on the cost, and thus the c_{err} would be smaller.

4.4.1 *Cardinality-Slope Robustness Metric*

To define a robustness metric that can be used at query optimization time, we argue that the following design considerations have to be taken into account:

1. The effort to calculate a robustness value.
2. The varying uncertainty of cardinality estimations for different types of operators.
3. The potential propagation of cardinality estimation errors.

A low calculation effort is essential to use a metric at query optimization time. The risk of cardinality estimation errors for different types of operators has to be considered in the robustness metric, because it has been shown that the precision of statistical models varies for different types of operators [33, 73, 78]. Finally, it has to be considered that cardinality estimation errors on deep edges, i.e., greater depth in the query execution plan tree [57], can be propagated to the cardinality estimations on higher edges, i.e., smaller depth in the plan tree. Consequently, cardinality estimation errors on deep edges can have a stronger impact on c_{err} compared to higher edges.

To describe the cardinality-slope robustness metric, we denote a query execution plan $P = (O_P, E_P)$, where O_P is the set of operators and E_P the set of edges. We take the PCFs for all edges in a query execution plan into account. The next building block for the cardinality-slope robustness metric is the definition of a *cardinality-slope value* for an edge $e \in E_P$ based on a PCF of cardinality f on e.

Definition 4.4. *The* ***cardinality-slope value*** $\delta_{f,e}$ *for an edge* $e \in E_P$ *is the slope of* $PCF_{f,e}$ *at the estimated cardinality* $\hat{f}$*, where* $PCF_{f,e}$ *is the PCF that models the cost of a plan* P *as a function of cardinality* f *on* e*.*

In theory, estimation errors can occur on all edges in the query execution plan. In practice, the precision of statistical models for cardinality estimation varies for different types of operators. For example,

edges after foreign key joins can be estimated more precisely than after $m{:}n$ joins, due to the constraint on keys [33, 78]. Also, edges after base table scans can be estimated more precisely than edges after filter predicates. To consider the different risks for estimation errors, we define an *edge weighting function* φ as the next building block for the cardinality-slope robustness metric.

Definition 4.5. *An **edge weighting function** $\varphi : E_P \to [0.0, 1.0]$ assigns each edge $e \in E_P$ an estimation-uncertainty value between 0.0 (not sensitive) and 1.0 (very sensitive).*

An edge after a $m{:}n$ join should get a larger error-sensitivity value, e.g., 1.0, compared to an edge after a foreign key join , e.g., 0.0. The following definition combines the building blocks to a metric.

Definition 4.6. *The robustness value r_{δ_f} of the **cardinality-slope robustness metric** for a plan P is defined as the sum over the products of $\delta_{f,e}$ and $\varphi(e)$ for each edge $e \in E_P$:*

$$r_{\delta_f}(P) = \sum_{e \in E_P} \varphi(e) \cdot \delta_{f,e}$$

Consequently, the plan is the more robust, the smaller the robustness value is. In Section 4.6, we experimentally evaluate the cardinality-slope robustness metric with respect to the consistency requirements of Section 4.2. The cardinality-slope robustness metric also follows our design considerations: Our experiments in Section 4.6 expose the low calculation overhead for r_{δ_f}. Potential cardinality estimation errors for different types of operators are weighted by φ. Finally, Definition 4.6 implicitly considers the potential propagation of cardinality estimation errors. As Theorem 4.1 reveals, cardinality estimation errors on deep edges in query execution plans can have a stronger impact on the total cost and therefore the robustness value r_{δ_f}, compared to higher edges. This is not the case, when there is a very selective operator between the deep and the higher edge: a very selective operator can decrease the number of output tuples to almost zero. Consequently, the cardinality estimation errors in the underlying sub-plan have almost no impact on the total cost, and therefore on the robustness value r_{δ_f} anymore. We formalize and prove this observation in Theorem 4.1.

Theorem 4.1. *Assuming a deep plan edge i with estimated cardinality $\hat{f}_i$ and cardinality-slope value $\delta_{f,i}$, and a higher plan edge j with estimated cardinality $\hat{f}_j$ and cardinality-slope value $\delta_{f,j}$. Then, for C_{out} it holds:*

$$\delta_{f,i} \geqslant \delta_{f,j} \Leftrightarrow \frac{\hat{f}_j}{\hat{f}_i} \geqslant \frac{\delta_{f,j} - S}{1 + \delta_{f,j}}, \tag{4.3}$$

where $S \geqslant 0$ depends on the estimated cardinalities and selectivities between the deep edge i and the higher edge j.

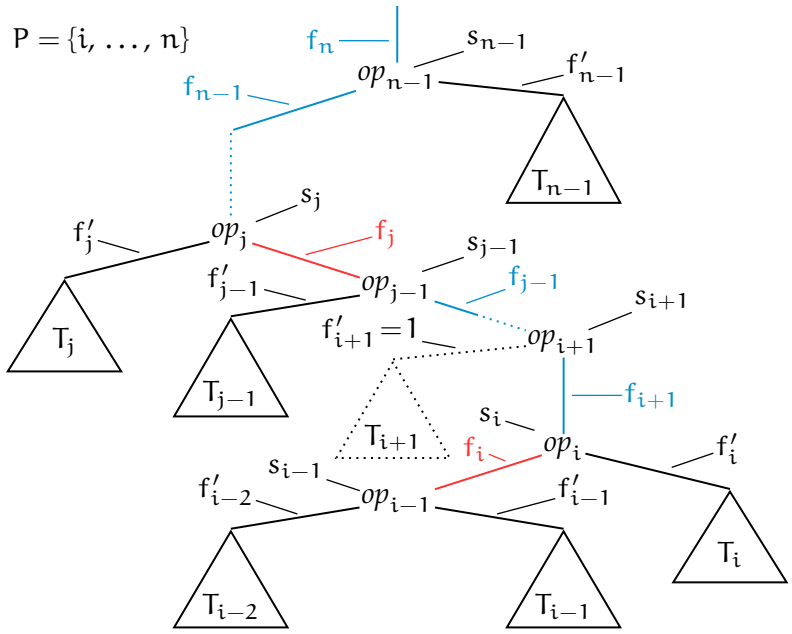


Figure 4.6: Arbitrary path in a query execution plan.

Proof. Figure 4.6 shows an arbitrary path P from edge i over edge j to the root edge n, in an arbitrary query execution plan. The arbitrary execution plan can have arbitrary operators and an arbitrary tree structure. The path contains unary and binary operators. We denote the cardinality of an edge e as f_e, and the selectivity of an operator op as s_{op}. It also shows non-path edges $e' \notin P$ with an arbitrary sub-tree T_e and cardinality f'_e. The cardinality of an edge $e \in P \setminus \{i\}$ is defined as $f_e = f_{e-1} \cdot f'_{e-1} \cdot s_{e-1}$. For a more convenient notation we define for edge i that $f_i = f'_{i-1} \cdot f'_{i-2} \cdot s_{i-1}$. For unary operators such as filters, there is only one input edge, and therefore we add without loss of generality a second invisible input edge with cardinality 1 (see f'_{i+1} in Figure 4.6). Therefore, we can rewrite the estimated cardinality on an edge $e \in P$:

$$\hat{f}_e = \prod_{k=i-2}^{e-1} \hat{f}'_k \cdot \prod_{k=i-1}^{e-1} \hat{s}_k \tag{4.4}$$

We assume an edge $j \in P$ (see Figure 4.6) such that $i < j$, i.e., the edge i is deeper than the edge j. To see the impact of the estimated cardinality of the deeper edge $\hat{f}_i$, we rewrite the estimated cardinality of the higher edge $\hat{f}_j$ as:

$$\hat{f}_j = \hat{f}_i \cdot \prod_{k=i}^{j-1} \hat{f}'_k \cdot \prod_{k=i}^{j-1} \hat{s}_k = \hat{f}_i \cdot X \tag{4.5}$$

Before using Equation 4.5, we use Equation 4.4 to rewrite the C_{out} cost function. C_{out} is the sum over the estimated cardinalities of all

edges, i.e., all edges $e \in P$, and all other edges $e' \notin P$ including all edges from their sub-tree T_e.

$$C_{out} = \sum_{l=i}^{n} \left(\prod_{k=i-2}^{l-1} \hat{f}'_k \cdot \prod_{k=i-1}^{l-1} \hat{s}_k \right) + \sum_{k=i-2}^{n-1} \hat{f}'_k + \sum_{k=i-2}^{n-1} C_{out}(T_k) \tag{4.6}$$

Next, we construct a PCF that models the C_{out} costs as a function of f_i, i.e., $PCF_{f,i}$. To do so, we factor out $\hat{f}_i$ from Equation 4.6 and use variable f_i instead of the estimation $\hat{f}_i$.

$$C_{out} = f_i \cdot \underbrace{\left[\sum_{l=i}^{n} \left(\prod_{k=i}^{l-1} \hat{f}'_k \cdot \prod_{k=i}^{l-1} \hat{s}_k \right) \right]}_{\text{dependent on } f_i \ (\delta_{f,i})} + \underbrace{\sum_{k=i-2}^{n-1} \hat{f}'_k + \sum_{k=i-2}^{n-1} C_{out}(T_k)}_{\text{independent of } f_i \ (c_{const,i})} \tag{4.7}$$

We observe from Equation 4.7 that the costs dependent on f_i are the cardinality-slope value $\delta_{f,i}$ for the edge i. Next, we construct the $PCF_{f,j}$ for the higher edge j. We also separate the sum over the edges of P from Equation 4.6 into edges higher and deeper than j.

$$C_{out} = f_j \cdot \underbrace{\left[\sum_{l=j}^{n} \left(\prod_{k=j}^{l-1} \hat{f}'_k \cdot \prod_{k=j}^{l-1} \hat{s}_k \right) \right]}_{\text{dependent on } f_j \ (\delta_{f,j})} + \underbrace{\sum_{l=i}^{j-1} \hat{f}_l + \sum_{k=i-2}^{n-1} \hat{f}'_k + \sum_{k=i-2}^{n-1} C_{out}(T_k)}_{\text{independent of } f_j \ (c_{const,j})} \tag{4.8}$$

Now, we reformulate the part of Equation 4.7 that depends on f_i, to quantify the impact of $\delta_{f,j}$ on $\delta_{f,i}$. We separate the sums into one running from i to $j-1$ and another from j to n, and factor out X (see Equation 4.5) from the latter sum.

$$\delta_{f,i} = \underbrace{\sum_{l=i}^{j-1} \left(\prod_{k=i}^{l-1} \hat{f}'_k \cdot \prod_{k=i}^{l-1} \hat{s}_k \right)}_{S+X} + \underbrace{\prod_{k=i}^{j-1} \hat{f}'_k \cdot \prod_{k=i}^{j-1} \hat{s}_k}_{X} \cdot \underbrace{\left[\sum_{l=j}^{n} \left(\prod_{k=j}^{l-1} \hat{f}'_k \cdot \prod_{k=j}^{l-1} \hat{s}_k \right) \right]}_{\delta_{f,j}} \tag{4.9}$$

We observe that the product, denoted as X, is the last term of $S + X$. Let us presume $\delta_{f,i} \geqslant \delta_{f,j}$:

$$\delta_{f,i} \geqslant \delta_{f,j} \overset{(4.9)}{\iff} S + X \cdot (1 + \delta_{f,j}) \geqslant \delta_{f,j} \tag{4.10}$$

$$\iff X \geqslant \frac{\delta_{f,j} - S}{1 + \delta_{f,j}} \overset{(4.5)}{\iff} \frac{\hat{f}_j}{\hat{f}_i} \geqslant \frac{\delta_{f,j} - S}{1 + \delta_{f,j}} \tag{4.11}$$

By inserting $\delta_{f,i}$ into Equation 4.10, we factored out X. From Equation 4.10 to 4.11, we first subtracted S and second divided it by $1 + \delta_{f,j}$. Finally, Equation 4.5 is inserted. □

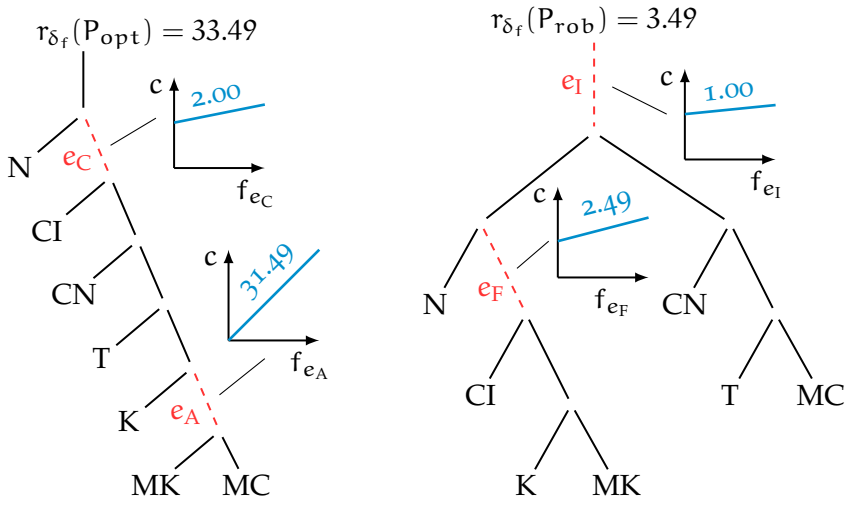
From Theorem 4.1, we observe that the right-hand side term of Equation 4.3 is always less than 1, since cardinalities and selectivities

are non-negative. Therefore, if the estimated cardinality of the deep edge is smaller or equal to the estimated cardinality of the higher edge ($\hat{f}_j / \hat{f}_i \geqslant 1$), then the deep edge has a larger cardinality-slope value ($\delta_{f,i} > \delta_{f,j}$). In contrast, if there is a highly selective operator between the deep and the higher edge ($\hat{f}_j / \hat{f}_i < 1$), then the right-hand side term of Equation 4.3 is a tight bound for $\delta_{f,i} \geqslant \delta_{f,j}$, i.e., for a highly selective operator $\delta_{f,i} < \delta_{f,j}$ can hold. Furthermore, Theorem 4.1 can be extended and proven for cost functions other than C_{out}. The reason is that cost functions in general have dependencies on cardinalities, and the cardinality of an edge is always a parameter in the following cardinality estimations towards the root.

Let us again consider Query 17 of the Join Order Benchmark. Figure 4.7 shows the robustness value calculation of the cardinality-slope robustness metric for the estimated optimal plan P_{opt} and the estimated most robust plan P_{rob}, i.e., the plan with the minimum robustness value r_{δ_f} from the robust plan candidates. For simplicity, we use an edge weighting function φ that assigns weight 1.0 to all $m{:}n$ join edges and weight 0.0 to all foreign key and base table scan edges. For both plans, the dashed edges are the edges that include $m{:}n$ joins, i.e., the edges that are more sensitive to estimation errors. The corresponding PCF, including the δ_f value, is shown to the right of those edges. For example, the cardinality-slope value δ_{f,e_F} for edge e_F of P_{rob} is 2.49, i.e., the slope of PCF_{f,e_F} as calculated in Equation 4.1. As a result, the robustness value r_{δ_f} for $P_{opt} = 33.49$ and for $P_{rob} = 3.49$. Therefore, P_{rob} is more robust according to the cardinality-slope metric than P_{opt}. The true statistics in the Figures 4.3 and 4.4 result in a smaller c_{err} for P_{rob} than for P_{opt}. Executing both plans on the real-world database of the Join Order Benchmark shows a query execution time speedup of factor two for P_{rob} compared to P_{opt}.

4.4.2 *Selectivity-Slope Robustness Metric*

The cardinality-slope value δ_f for an edge $e \in E_P$ expresses the impact of one additional tuple on the total cost. Apart from the edge weighting function φ for potential cardinality estimation errors for different types of operators and the implicitly considered propagation of cardinality estimation errors, the edges in r_{δ_f} are not further weighted. In order to explain a derived robustness metric, we first denote $\hat{f}_{max}$ as the estimated maximum output cardinality of an operator. Taking a binary join as an example, $\hat{f}_{max}$ is the product of its estimated input cardinalities (cross-product). We argue that edges with a potentially larger absolute cardinality error Δf, i.e., the absolute difference between the estimated and the true cardinality, can have a stronger impact on the final c_{err}. Since Δf cannot be calculated before plan execution, the next robustness metric considers the risk of a large Δf, by taking $\hat{f}_{max}$ into account. The larger $\hat{f}_{max}$ is, the larger



(a) Estimated Optimal Plan (b) Estimated Robust Plan

Figure 4.7: Robustness values r_{δ_f} assigned to robust plan candidates for JOB Query 17.

is the potential impact on the final c_{err}. Next, we define the *selectivity-slope value* δ_s and the corresponding *selectivity-slope robustness metric*.

Definition 4.7. *The* ***selectivity-slope value*** $\delta_{s,op}$ *for an operator* $op \in O_P$ *is the slope of* $PCF_{s,op}$ *at the estimated selectivity* $\hat{s}$*, where* $PCF_{s,op}$ *is the PCF that models the cost for a plan* P *as a function of the selectivity* s *on* op*.*

Definition 4.8. *The robustness value* r_{δ_s} *of the* ***selectivity-slope robustness metric*** *for a plan* P *is the sum over the products of* $\delta_{s,op}$ *and* $\phi(op)$ *for each operator* $op \in O_P$*:*

$$r_{\delta_s}(P) = \sum_{op \in O_P} \phi(op) \cdot \delta_{s,op},$$

where $\phi : O_P \to [0.0, 1.0]$ *is a weighting function for operators, instead of edges as* φ*, to reflect the uncertainty of cardinality estimations.*

After defining the selectivity-slope robustness metric, we show that the metric implicitly weights the cardinality-slope value $\delta_{f,e}$ for the outgoing edge $e \in E_P$ of an operator $op \in O_P$ by $\hat{f}_{max}$.

Theorem 4.2. *For* C_{out}*, the selectivity-slope value* $\delta_{s,op}$ *of an operator* $op \in O_P$ *is the product of* $\hat{f}_{max}$ *and* $\delta_{f,e}$ *on the outgoing edge* $e \in E_P$ *of* op*.*

Proof. Without loss of generality, we consider the edge i in Figure 4.6 with the cardinality f_i. From Equation 4.7 in the proof of Theorem 4.1, we see that the $PCF_{f,i}$ for edge i consists of costs independent of f_i, $c_{const,i}$, and costs dependent on f_i, i.e., the cardinality-slope value $\delta_{f,i}$.

$$C_{out} = f_i \cdot \delta_{f,i} + c_{const,i} \tag{4.12}$$

As in Figure 4.6, we denote s_{i-1} as the selectivity of operator op_{i-1}, i.e., the operator before edge i. Furthermore, we denote f'_{i-1} and f'_{i-2} as the input cardinalities of op_{i-1}. The cardinality of edge i is the product of both input cardinalities of op_{i-1} and the selectivity s_{i-1}, i.e., $f_i = f'_{i-1} \cdot f'_{i-2} \cdot s_{i-1}$. For unary operators, which have only one input edge, such as filters, we added in Theorem 4.1 without loss of generality a second invisible input edge with cardinality 1 (see f'_{i+1} in Figure 4.6). Therefore, we rewrite Equation 4.12 as:

$$C_{out} = f'_{i-1} \cdot f'_{i-2} \cdot s_{i-1} \cdot \delta_{f,i} + c_{const,i} \tag{4.13}$$

To rewrite Equation 4.13 into a PCF with s_{i-1} as a single cost parameter ($PCF_{s,i-1}$), the cardinality variables for the input edges f'_{i-1} and f'_{i-2} are set to the corresponding estimates $\hat{f}'_{i-1}$ and $\hat{f}'_{i-2}$.

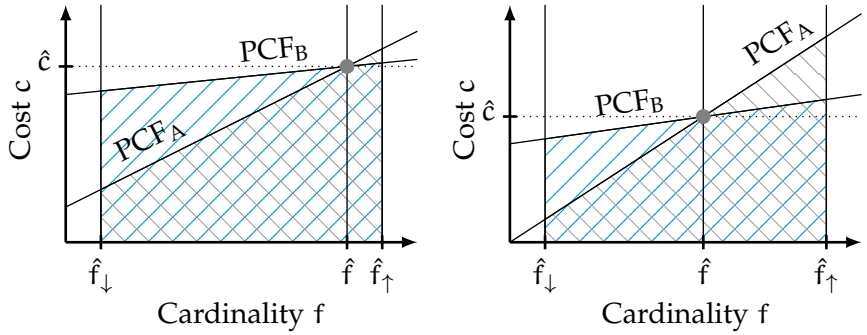
$$C_{out} = \hat{f}'_{i-1} \cdot \hat{f}'_{i-2} \cdot \delta_{f,i} \cdot s_{i-1} + c_{const,i} = \delta_{s,i-1} \cdot s_{i-1} + c_{const,i} \tag{4.14}$$

Since $c_{const,i}$ is independent of f_i, it has no cost depending on s_{i-1}. Therefore, $\delta_{s,i-1}$ for the operator op_{i-1} is the product of $\hat{f}_{max} = \hat{f}'_{i-1} \cdot \hat{f}'_{i-2}$, and the cardinality-slope value $\delta_{f,i}$ of the outgoing edge i of the operator op_{i-1}. Note that unary operators have only one input edge, so that $\hat{f}'_{i-2}$ or $\hat{f}'_{i-1}$ is set to 1, and therefore has no impact. □

In Section 4.6, we experimentally evaluate the selectivity-slope robustness metric with respect to the consistency requirements of Section 4.2. The selectivity-slope robustness metric also follows our design considerations: the calculation effort is small, potential cardinality estimation errors are weighted, and the propagation of cardinality estimation is considered. A proof for the latter can be constructed analogous to the proof of Theorem 4.1 by adding the additional weight of $\hat{f}_{max}$. In summary, the selectivity-slope robustness metric additionally considers the risk of a large Δf on all edges, compared to the cardinality-slope robustness metric.

4.4.3 *Cardinality-Integral Robustness Metric*

The next robustness metric is a trade-off between plan robustness and estimated costs. Both the cardinality-slope and the selectivity-slope robustness metric use the slopes of PCFs as the robustness indicator. However, a plan with a steep slope could still have smaller costs for a significant range of cardinality values, compared to a plan with a more moderate slope. Figure 4.8a shows PCF_A and PCF_B of two different plans as a function of cardinality on a single plan edge. The cardinality of the edge is denoted on the x-axis and the cost on the y-axis. Furthermore, it shows $\hat{f}$, $\hat{f}_\downarrow$, and $\hat{f}_\uparrow$, where $\hat{f}_\downarrow$ is the lower bound for the estimated cardinality of an edge $e \in E_P$, and $\hat{f}_\uparrow$ is the upper bound for the estimated cardinality of $e \in E_P$. We argue that a lower and an upper bound for the estimated cardinality can make



(a) Considerable integral difference. (b) Marginal integral difference.

Figure 4.8: Conceptual comparison between slope and integral indicator.

the robustness metric more precise. In practice, histograms, sampling, or bounds for cardinality estimation [58] can give estimations for $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$. In the evaluation in Section 4.6, we set $\hat{f}_\downarrow$ to 0, and $\hat{f}_\uparrow$ to $\hat{f}_{max}$. Note that PCF_A and PCF_B have the same estimated costs $\hat{c}$ at $\hat{f}$. Since PCF_B has a more moderate slope than PCF_A, the cardinality-slope robustness metric would assign PCF_B a smaller robustness value than PCF_A. By considering the costs between $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$, a metric that is a trade-off between plan robustness and cost would assign PCF_A a better value compared to PCF_B. The reason is that PCF_A has significant less cost for a majority of cardinalities between $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$, i.e., PCF_A has significantly less cost between $\hat{f}_\downarrow$ and $\hat{f}$ than PCF_B, and is competitive to PCF_B between $\hat{f}$ and $\hat{f}_\uparrow$. To model plan robustness and cost in a single value, we consider the integral of the PCF between $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$. In Figure 4.8a, the integral of PCF_A is smaller than the integral of PCF_B. Next, we define the *cardinality-integral value* $\int_f$ as a trade-off between plan robustness and cost, and afterwards the *cardinality-integral robustness metric*.

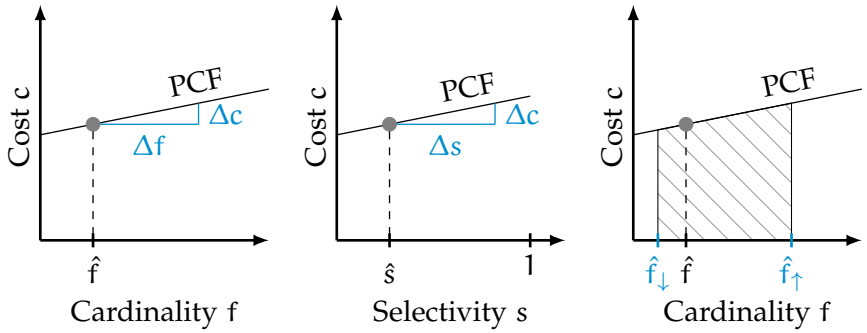
Definition 4.9. *The* ***cardinality-integral value*** $\int_{f,e}$ *for an edge* e *is:*

$$\int_{\hat{f}_\downarrow}^{\hat{f}_\uparrow} PCF_{f,e}$$

Definition 4.10. *The robustness value* $r_{\int_f}$ *of the* ***cardinality-integral robustness metric*** *for a plan* P *is defined as the sum over the products of* $\int_{f,e}$ *and* $\varphi(e)$ *for each edge* $e \in E_P$*:*

$$r_{\int_f}(P) = \sum_{e \in E} \varphi(e) \cdot \int_{f,e}$$

A second scenario in Figure 4.8b shows the impact of the lower and upper bound $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$. In Figure 4.8b, the integral between $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$ of PCF_A is only slightly smaller, than the integral of PCF_B. Consequently, the cardinality-integral robustness metric assigns PCF_A a



(a) Cardinality-Slope (b) Selectivity-Slope (c) Cardinality-Integral

Figure 4.9: Overview of Cardinality-slope, selectivity-slope and cardinality-integral robustness metric.

better value compared to PCF_B, although PCF_B has the more robust cost behavior. Note that the cardinality-slope robustness metric assigns PCF_B a better robustness value than PCF_A, because of the more moderate slope of PCF_B. Both scenarios in Figure 4.8 show that the choice of a lower and an upper bound, $\hat{f}_\downarrow$ and $\hat{f}_\uparrow$, for the estimated cardinality of an edge $e \in E_P$ has a considerable impact on the cardinality integral robustness metric.

Calculating the integral makes the metric independent of $\hat{f}$ and the slope at this point. In addition, we can support arbitrary PCF shapes, because integrals can always be approximated numerically [8]. Section 4.6 shows the experimental evaluation of the cardinality-integral robustness metric with respect to the consistency requirements of Section 4.2. The cardinality-integral metric follows two design considerations: It has a low calculation effort, and potential cardinality estimation errors are weighted. Since the cardinality-integral metric calculates integrals to balance plan robustness and costs, it considers high plan edges stronger than deeper plan edges. This is because plan edges always contain the cost of their sub-plans. Consequently, the integrals are larger on high plan edges compared to the deeper plan edges, and therefore have a higher impact on the robustness value.

4.4.4 *Robustness Metrics Overview*

Figure 4.9 summarizes the three robustness metrics. The cardinality-slope robustness metric (Figure 4.9a) reflects the expected difference between estimated and true cost for cardinality estimation errors on all edges in the query execution plan. Furthermore, it implicitly considers the potential propagation of cardinality estimation errors, and takes the potential cardinality estimation errors for different types of operators into account. In addition, the selectivity-slope robustness metric (Figure 4.9b) considers the risk of a large absolute cardinality error Δf on all edges. Therefore, it models the PCFs as a func-

tion of operator selectivity. In contrast to the cardinality-slope and the selectivity-slope robustness metric, the cardinality-integral metric (Figure 4.9c) does not purely focus on plan robustness, but takes also costs into account. Furthermore, it can consider a more realistic range for the cardinality of an edge. All three metrics support any kind of operator, operator implementation and query execution plan trees, because the cost of a plan can always be modeled as a PCF of cardinality. In addition, the metrics can be extended to consider estimation errors in other cost parameters, such as consumed memory. We also experimented with a fourth metric, namely *selectivity-integral*, but found no substantial improvement over the cardinality-integral metric.

4.5 ROBUST PLAN CANDIDATES AND ROBUST PLAN SELECTION

Our novel robust plan selection strategy has three phases: First, we enumerate the set of *robust plan candidates*. Every robust plan candidate is a plan for the entire query, and not a sub-plan. Second, we calculate the robustness value for each robust plan candidate by applying one of the three robustness metrics. Third, we select the *estimated most robust plan*, i.e., the robust plan candidate with the smallest robustness value for execution. Apart from robustness, selecting a cheap query execution plan is still a major optimization goal. Consequently, our first criterion for the robust plan candidates is that they have to be the *k-cheapest plans*:

Definition 4.11. *The* **k-*cheapest plans*** *are the* k *query execution plans with the smallest estimated cost.*

The *k*-cheapest plans significantly reduce the number of plan candidates, and give a tight upper bound for the number of plans independent of the plan space. In addition, the *k*-cheapest plans can be utilized to apply additional constraints, such as memory consumption. The experiments in Section 4.6 reveal that $k = 500$ results in a low optimization overhead. We further show that the estimated most robust plan inside $k = 500$ is competitive with respect to an estimated most robust plan with a larger k. Enumerating the *k*-cheapest plans is just a small modification in the query execution plan enumerator. The trivial approach in a dynamic programming enumerator is to keep the *k*-cheapest plans in each plan class, instead of only the cheapest plan. The *k*-cheapest plans of two plan classes can be combined to create plans of another plan class. We show in Section 4.6 that enumerating the *k*-cheapest plans has a reasonable overhead.

The *k*-cheapest plans can contain expensive plans for small queries. Only the cardinality-integral robustness metric takes plan cost into consideration. Therefore, we further limit the robust plan candidates, after being completely enumerated, for the cardinality-slope and the selectivity-slope robustness metric to near-optimal plans:

Definition 4.12. *The **near-optimal plans** are a sub-set of the plan space, containing the query execution plans with estimated cost at most λ-times larger than the estimated cost of the estimated optimal plan.*

The near-optimal plans guarantee that robust plan candidates are competitive to the estimated optimal plan, i.e., the estimated cheapest plan. Furthermore, λ gives a theoretical upper bound for the increase of estimated cost of the robust plan compared to the optimal plan. Abhirama et al. [59] argue for $\lambda = 1.2$, which we confirm in Section 4.6.

In sum, our plan selection strategy has very low risks: First, we enumerate the *k*-cheapest plans. Second, we calculate the robustness value for each robust plan candidate. Though it is a reasonable overhead, it can be significant in very short running queries. It is not significant in our real-world experiments in Section 4.6.1. In addition, dynamic programming enumeration is no limitation, but shows that our approach can be integrated into enterprise class optimizers.

4.6 EXPERIMENTAL EVALUATION

We implemented the three robustness metrics in our dynamic programming join optimizer (see Section 2.7). We use the same optimizer to determine the baseline plan for each query, i.e., the estimated cheapest or optimal plan. Our join optimizer relies on dynamic programming [4], such as DB2 [17] and Postgres [78]. As Postgres, it exhaustively searches the plan space including bushy trees. We show in Section 2.7 that its cardinality estimator is competitive. In addition, we use the C_{mm} [93] cost function, which is an extension of C_{out} [26] that considers different operator types and operator implementations. C_{mm} has a strong correlation to our main-memory execution engine, as we show in Section 2.7. Taking everything together, we argue that our join optimizer's choice of the estimated optimal plan is very similar to the choice of popular commercial and free systems, for the considered join queries. We denote its estimated optimal plan choice as conventional plan, and consider it as the baseline. We use our main-memory execution engine to determine query execution times.

We experimentally evaluate the plan selection strategies with respect to their end-to-end query execution times in Section 4.6.1, and plan robustness in Section 4.6.2. The results in Section 4.6.2 do not depend on the machine the experiments run on. Reported execution times were taken on a two socket Intel Xeon E5-2660 v3 system with 128 GB of main memory, running a Linux 4.4.120 kernel. As we explain in Section 2.7, our main-memory query execution engine performs join operators as hash joins. The query optimizer and metric implementations are single-threaded. The entire system is compiled with `gcc` version 7.2.0 and optimization option -O3.

Our first workload is based on the Join Order Benchmark (JOB) [78]. The JOB uses the real-world database from IMDb with skew, correla-

tions, and different join relationships that cause estimation errors. We modified the original queries to be pure join queries, which results in 33 complex queries containing cycles and multiple join conditions between sub-plans. Since pure join queries without any filters on base tables create large intermediate results, we use the `movie_id` column as scale factor for the benchmark. We limit the `movie_id` column of all tables to values less equal than $100,000$. In the end, the scale factor enables to run 31 different queries. We argue that the scale factor does not limit the validity of our results, because it creates a snapshot of the database at the point when it contained only $100,000$ movies.

Our second benchmark is synthetic with generated data and join queries. The query topologies are: chain, cycle, star, and snowflake. All topologies join 10 tables. The snowflake topology has a fact table with three dimension tables. Each dimension table has again two sub-dimensions. For each topology, we create one query and 100 different data sets. Furthermore, we generate 100 queries with a random topology and a corresponding data set. The random topology creator starts with a random connected query graph, and adds with a probability of 4% additional edges to create cycles. The random topologies also join 10 tables. For all generated data sets, the base table cardinalities are uniform random numbers between $10,000$ and $100,000$. The data sets contain skew and arbitrary correlations between columns to generate expanding and selective joins. There are foreign key and $m{:}n$ join relationships. The join cardinalities between two base tables R_i and R_j are uniform random numbers between $max(|R_i|,|R_j|) - 5000$ and $max(|R_i|,|R_j|) + 5000$. We use exactly the same setup, i.e., system, kernel, compiler, and benchmarks, for the experiments in Chapter 5.

Each experiment starts with enumerating the robust plan candidates. For the cardinality-slope and the selectivity-slope metric, the robust plan candidates are defined by the near-optimal plans ($\lambda = 1.2$) and the k-cheapest plans ($k = 500$). For the cardinality-integral metric it is only the k-cheapest plans ($k = 500$). By definition, the k-cheapest plans contain the estimated optimal plan, which is the baseline in our experiments. To select the estimated most robust plan, each metric assigns a robustness value to every robust plan candidate. Both workloads contain only join queries with at least one $m{:}n$ join, and there are no estimation errors for foreign key joins and base table scans in our setup. Therefore, we define the weighting functions φ and ϕ to be 1.0 for $m{:}n$ joins, and 0.0 for foreign key joins and base table scans.

We compare our baseline, the estimated optimal plan (EO), with the estimated most robust plan according to one of the metrics: cardinality-slope (FS), selectivity-slope (SS), and cardinality-integral metric (FI). We also perform a best-case offline analysis to show the potential of robust plan selection. We execute all robust plan candidates and denote the plan with the shortest execution time as the fastest plan (FA).

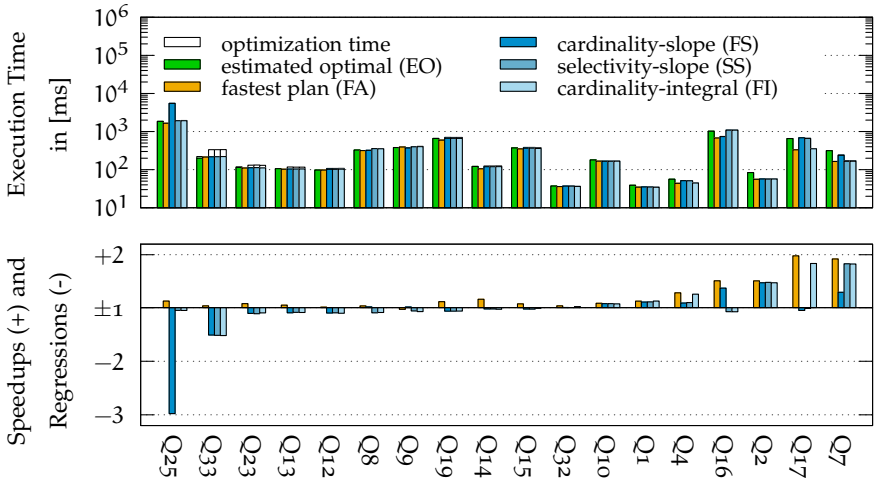


Figure 4.10: Typical results of experiment comparing the end-to-end query execution times of the estimated optimal query execution plan (EO) and fastest query execution plan (FA) with the estimated most robust query execution plans according to one of the three robustness metrics (FS, SS, and FI) on different Join Order Benchmark queries.

4.6.1 *Query Execution Time*

In the initial experiment, we demonstrate that the estimated most robust query execution plans have on average a faster end-to-end query execution time, compared to the estimated optimal or cheapest query execution plans. Figure 4.10 shows some typical results of Join Order Benchmark queries, plotted along the x-axis. The y-axis shows the median end-to-end query execution time t_P for a plan P in milliseconds over 101 executions in logarithmic scale. In addition, the y-axis shows the resulting speedup ($+\,t_{EO}/t_P$) or regression ($-\,t_P/t_{EO}$) of robust plan selection with respect to conventional plan selection denoted as EO. We order the queries on the x-axis by the speedup of robust plan selection compared to conventional plan selection. The typical results we show include the queries with the best speedup, e.g., Q2 and Q7, and the worst regression, e.g., Q25 and Q33. The figures in Section 4.6.2 show the same queries in the same order.

The best speedup is achieved for the cardinality-slope metric (FS) in Q2 (1.47), and for selectivity-slope metric (SS) and cardinality-integral metric (FI) in Q7 (1.83). In contrast, the worst regression for SS and FI is 1.52 (Q33). For FS the worst regression is 2.98 (Q25), but the second worst regression is again 1.51 (Q33). By considering near-optimal plans and the k-cheapest plans, the estimated most robust plan does not necessarily have the smallest estimated cost and small regressions for some queries can be the result. Comparing the results of Q2 and Q7 to the fastest plan FA, which was found in a brute-force analysis,

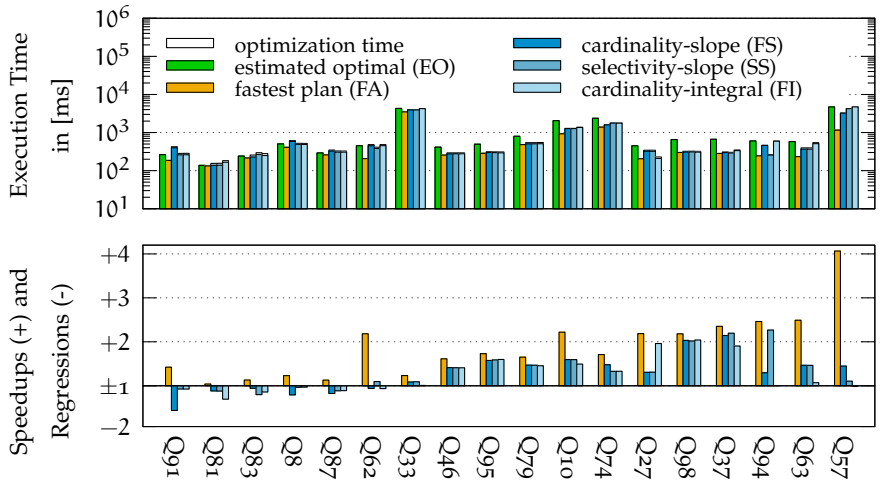


Figure 4.11: Typical results of experiment comparing the end-to-end query execution times of the estimated optimal query execution plan (EO) and fastest query execution plan (FA) with the estimated most robust query execution plans according to one of the three robustness metrics (FS, SS, and FI) on synthetic benchmark queries with random query topology.

shows that all robust plan selection strategies achieve similar query execution times as FA. Consequently, the probability is high that the robust plans are close to the true optimal plans in these cases.

Figure 4.11 shows some typical results for queries of the synthetic benchmark with a random query topology. The results include Q37, Q94, and Q98 with the best speedup as well as Q81, Q83, and Q91 with the worst regression. For all three metrics, the robust plan selection achieves a better cumulative query execution time than conventional plan selection, which executes the estimated optimal plan EO. Furthermore, all three metrics achieve larger speedup than regression factors. Comparing the results of Q37, Q95, and Q98 to the fastest plan FA, shows that all robust plan selection strategies achieve similar query execution times as FA. Therefore, the probability is high that the robust plans are close to the true optimal plans in these cases.

Table 4.1 summarizes the results of all benchmarks, by cumulative query execution time, best speedup, and worst regression with respect to EO of all queries in the corresponding benchmark. Table 4.1 also reveals whether there is on average a regression or a speedup. For queries with chain, cycle, and random topology, the three robustness metrics achieve stronger speedups than regressions, and also on average, there is a speedup compared to the estimated optimal plans. For the Join Order Benchmark, star, and snowflake queries, the robustness plan selection cannot improve the average query execution times compared to the estimated optimal plans. In the JOB, the cardinality-slope metric FS even causes an average regression of

		Σ time	best speedup	worst regression	average
JOB	EO	13892 ms	–	–	–
	FS	17696 ms	+1.47×	−2.98×	−1.27×
	SS	14581 ms	+1.83×	−1.52×	−1.05×
	FI	14243 ms	+1.83×	−1.52×	−1.03×
	FA	12483 ms	+1.98×	–	+1.11×
Chain	EO	18798 ms	–	–	–
	FS	16091 ms	+3.31×	−1.26×	+1.17×
	SS	17061 ms	+1.79×	−1.36×	+1.10×
	FI	16865 ms	+3.49×	−1.13×	+1.11×
	FA	14562 ms	+4.23×	–	+1.29×
Cycle	EO	41084 ms	–	–	–
	FS	34587 ms	+2.43×	−1.21×	+1.19×
	SS	32279 ms	+2.43×	−1.27×	+1.27×
	FI	33193 ms	+2.43×	−1.25×	+1.24×
	FA	25539 ms	+2.94×	–	+1.61×
Star	EO	178520 ms	–	–	–
	FS	182693 ms	+1.14×	−1.49×	−1.02×
	SS	177251 ms	+1.23×	−1.48×	+1.01×
	FI	188334 ms	+1.21×	−1.48×	−1.05×
	FA	161987 ms	+1.47×	–	+1.10×
Snowflake	EO	53793 ms	–	–	–
	FS	54437 ms	+1.53×	−2.07×	−1.01×
	SS	51579 ms	+1.91×	−1.36×	+1.04×
	FI	53327 ms	+1.78×	−1.38×	+1.01×
	FA	44843 ms	+2.11×	–	+1.19×
Random	EO	82001 ms	–	–	–
	FS	75644 ms	+2.14×	−1.60×	+1.08×
	SS	74951 ms	+2.27×	−1.22×	+1.09×
	FI	78922 ms	+2.04×	−1.33×	+1.04×
	FA	56273 ms	+4.07×	–	+1.46×

Table 4.1: Summary of end-to-end query execution time comparisons between the estimated optimal plans (EO), the most robust plans according to cardinality-slope (FS), selectivity-slope (SS), or cardinality-integral metric (FI), and the fastest plans (FA) over all queries of different real-world and synthetic benchmarks.

27%. As we illustrate in Figure 4.10, the reason is Q25, for which the cardinality-slope metric FS fails. We further discuss the reasons why the cardinality-slope metric fails for Q25 in the following sections.

The root cause why the estimated most robust plans achieve in some benchmarks on average better query execution times compared to the estimated optimal plans are cardinality estimation errors. Conventional plan selection assumes that there are no cardinality estimation errors when selecting the estimated optimal, i.e., cheapest plan for execution. Since there are cardinality estimation errors in some of the queries, we consider this assumption does not hold and the estimated most robust plans perform better. If there are almost no estimation errors, selecting a robust plan cannot improve the query execution time, and small regressions can occur. In sum, the execution time experiments illustrate that all robustness metrics are effective.

4.6.2 *Plan Robustness*

After showing that our robust plan selection strategies achieve an average improvement of end-to-end query execution time, we next analyze some conceptual properties of our robustness metrics. In every subsection, we evaluate one of the robustness metric consistency requirements, we presented in Section 4.2.

4.6.2.1 *Cost Error Factor Improvement*

According to the first consistency requirement, the estimated most robust plan should have a smaller cost error factor c_{err} than the estimated optimal plan (see Section 4.2). To measure the cost error factor improvement, we calculate the difference between the c_{err} of the estimated optimal plan ($c_{err,EO}$) and the c_{err} of another plan P ($c_{err,P}$):

$$\Delta_{c_{err,P}} = c_{err,EO} - c_{err,P}$$

Consequently, a positive $\Delta_{c_{err,P}}$ reveals the c_{err} improvement of plan P compared to the estimated optimal plan EO. Figure 4.12 illustrates some typical results for the Join Order Benchmark. The x-axis shows the same queries in the same order as Figure 4.10. The $\Delta_{c_{err,P}}$ is denoted on the y-axis in logarithmic scale. The results include the queries with the largest $\Delta_{c_{err,P}}$, e.g., Q14, Q16, and Q17, and the smallest $\Delta_{c_{err,P}}$, e.g., Q9 and Q25. Our robust plan selection strategy using the cardinality-slope robustness metric FS achieves a positive $\Delta_{c_{err,P}}$ in 29 of 31 queries. There is a negative $\Delta_{c_{err,P}}$ of 0.03 in Q23, which is almost not visible in Figure 4.12, and a significant degradation of $\Delta_{c_{err,P}}$ in Q25. The negative $\Delta_{c_{err,P}}$ of Q25 also explains the corresponding end-to-end query execution time degradation in Figure 4.10. Our robust plan selection strategy using the selectivity-slope robustness metric SS and the cardinality-integral robustness metric FI has a negative $\Delta_{c_{err,P}}$ for Q9, but achieves a positive $\Delta_{c_{err,P}}$ in 30 of the 31 queries.

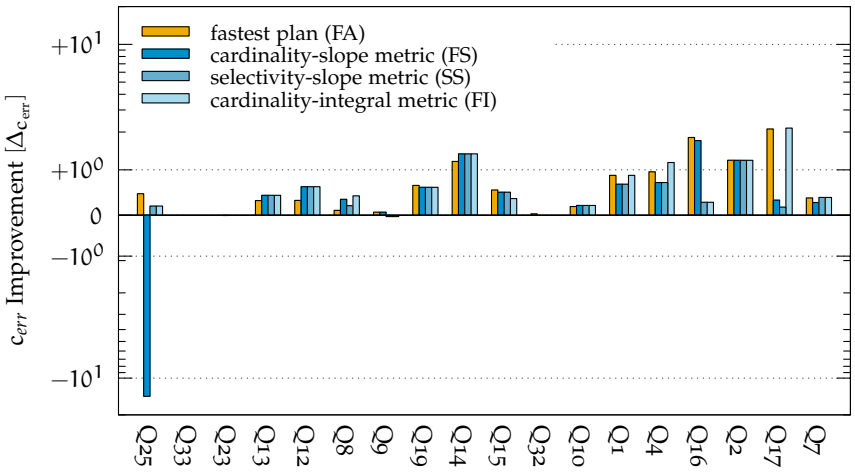


Figure 4.12: Typical results of experiment showing the c_{err} improvement with respect to the estimated optimal query execution plan, of the fastest query execution plan (FA), and the estimated most robust query execution plans according to one of the three metrics (FS, SS, and FI) on different Join Order Benchmark queries.

Comparing the $\Delta_{c_{err,P}}$ of the fastest plans FA to the estimated most robust plans shows that the fastest plans are not necessarily as robust as the estimated most robust plans, because there can be a considerable difference between estimated and true cost for FA. Considering, e.g., Q12 and Q14, reveals that our robust plan selection using the FS, SS, or FI robustness metrics results in a better $\Delta_{c_{err,P}}$ compared to FA.

Figure 4.13 illustrates some typical results of the synthetic benchmark queries with a random query topology. The queries are the same, and in the same order as in Figure 4.11. The query set includes the cases with the largest $\Delta_{c_{err,P}}$, e.g., Q57 and Q74, and the smallest $\Delta_{c_{err,P}}$, e.g., Q33, Q87, and Q91. Our robust plan selection using the cardinality-slope robustness metric FS achieves a positive $\Delta_{c_{err,P}}$ in 92 of 100 queries, the selectivity-slope robustness metric SS in 97 of 100 queries, and the cardinality-integral robustness metric FI in 88 of 100 queries. Comparing the $\Delta_{c_{err,P}}$ of the fastest plans FA to the estimated most robust plans shows that FA achieves in most cases the best $\Delta_{c_{err,P}}$. Consequently, the fastest plans in this benchmark are theoretically some of the most robust plans.

Table 4.2 summarizes the results of all benchmarks, by largest, smallest, and average $\Delta_{c_{err,P}}$ of all queries in the corresponding benchmark. Table 4.2 also contains the number of queries with a positive $\Delta_{c_{err,P}}$. Except for the cardinality-slope metric FS in the Join Order Benchmark, all three robustness metrics achieve on average a positive $\Delta_{c_{err,P}}$. Comparing the largest $\Delta_{c_{err}}$ with the smallest $\Delta_{c_{err}}$ of FS, SS and FI reveals that the maximum gains outweigh the maximum losses, except for the cardinality-slope metric FS in the Join Order Benchmark due to

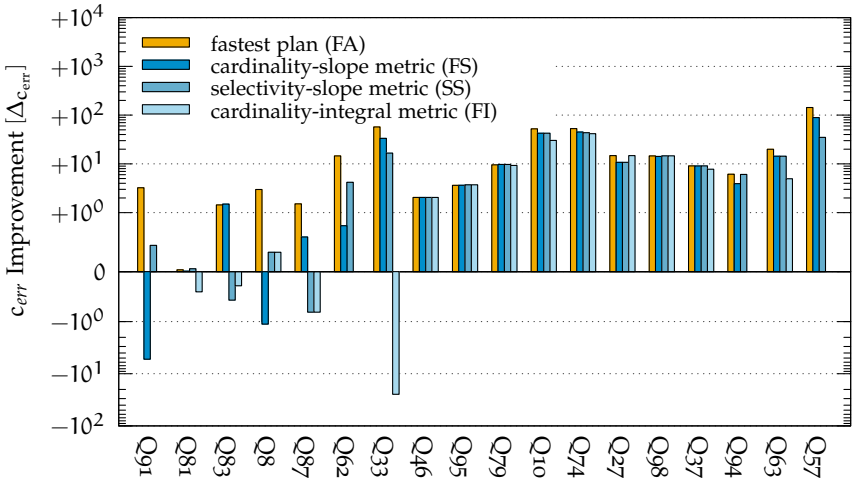


Figure 4.13: Typical results of experiment showing the c_{err} improvement with respect to the estimated optimal query execution plan, of the fastest query execution plan (FA), and the estimated most robust query execution plans according to one of the three robustness metrics (FS, SS, and FI) on synthetic benchmark queries with random query topology.

Q25. Considering FS and the random topology, shows the best $\Delta_{c_{err,P}}$ of a robustness metric with 89.02. This value results from Q57, which is illustrated in Figure 4.13. The comparison of FI to FS and SS in Table 4.2 shows that FS and SS achieve in most cases better results. The reason is that the cardinality-integral metric already balances plan robustness and cost. In contrast, the cardinality-slope and selectivity-slope metrics purely focus on plan robustness. The corresponding robust plan selection strategy considers cost with the near-optimal plans constraint. To sum it up, our experiments reveal that the robust plan selection using the robustness metrics improves the average cost error factor c_{err} of query execution plans with respect to conventional plan selection, which solely relies on estimated cost.

4.6.2.2 *Cost Error Factor Dominance*

According to the second consistency requirement, the estimated most robust plan, chosen by robust plan selection, should dominate all robust plan candidates, denoted as *RPC*, with respect to their c_{err} (see Section 4.2). In order to measure the cost error factor dominance, we define $\rho_{c_{err,P}}$ of a plan P:

$$\rho_{c_{err,P}} = \frac{|\{R | c_{err,P} \leqslant c_{err,R}, R \in RPC\}|}{RPC}$$

A $\rho_{c_{err,P}}$ of 100% indicates that a plan has the smallest c_{err} of all robust plan candidates, i.e., is the most robust plan. In practice, a $\rho_{c_{err,P}}$ of 100% is not achieved for every query, because the robustness

		largest $\Delta_{c_{err}}$	smallest $\Delta_{c_{err}}$	average $\Delta_{c_{err}}$	# pos $\Delta_{c_{err}}$
JOB	FS	+1.71	−14.10	−0.11	29/31
	SS	+1.34	−0.03	+0.29	30/31
	FI	+2.15	−0.03	+0.36	30/31
	FA	+2.12	0.00	+0.40	31/31
Chain	FS	+19.19	0.00	+0.80	98/100
	SS	+15.43	0.00	+0.73	100/100
	FI	+17.98	−0.20	+0.55	96/100
	FA	+20.69	0.00	+0.85	99/100
Cycle	FS	+25.04	0.00	+4.51	100/100
	SS	+24.87	0.00	+4.72	100/100
	FI	+20.02	−0.25	+3.30	98/100
	FA	+25.71	0.00	+5.62	100/100
Star	FS	+55.11	−0.70	+1.21	92/100
	SS	+48.37	−0.06	+1.34	98/100
	FI	+48.37	−6.96	+0.93	74/100
	FA	+67.70	−0.17	+1.79	96/100
Snowflake	FS	+18.07	−1.87	+1.28	93/100
	SS	+37.13	−2.87	+1.48	96/100
	FI	+13.47	−4.90	+0.75	93/100
	FA	+39.35	−2.03	+2.19	98/100
Random	FS	+89.02	−5.25	+5.86	92/100
	SS	+43.59	−0.81	+5.06	97/100
	FI	+41.64	−24.77	+2.12	88/100
	FA	+143.02	−0.096	+9.64	99/100

Table 4.2: Summary of cost error factor improvement comparisons between the most robust plans according to cardinality-slope (FS), selectivity-slope (SS), or cardinality-integral metric (FI), and the fastest plans (FA) relative to the estimated optimal plans (EO), over all queries of different real-world and synthetic benchmarks.

value assigned by a robustness metric is an approximation for an upper bound of c_{err}. Furthermore, there are cases in which the c_{err} values within *RPC* are close together, so that the expressiveness of $\rho_{c_{err,P}}$ is low. Therefore, we additionally define $\delta_{c_{err,P}}$ as the difference between the c_{err} of a plan P and the c_{err} of the most robust plan $R \in RPC$, i.e., the plan with the smallest c_{err}:

$$\delta_{c_{err,P}} = min\{c_{err,R} | R \in RPC\} - c_{err,P}$$

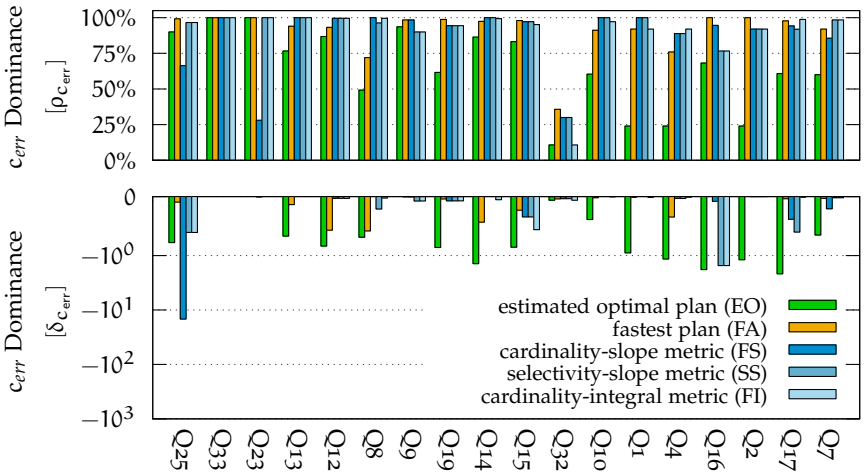


Figure 4.14: Typical results of experiment comparing the cost error factor dominance of the estimated optimal query execution plan (EO), the fastest query execution plan (FA), and the estimated most robust query execution plans according to one of the three metrics (FS, SS, and FI) on different Join Order Benchmark queries.

Consequently, the closer $\delta_{c_{err,P}}$ to 0 is, the better. A $\delta_{c_{err,P}}$ close to 0 indicates that a query execution plan P has a similar c_{err} as the query execution plan with the smallest c_{err} from the robust plan candidates, i.e., the most robust plan. Cases with small $\rho_{c_{err,P}}$ and in addition a $\delta_{c_{err,P}}$ close to 0 still indicate an almost optimal behavior. The considered plans per query in this analysis are the robust plan candidates, defined by the k-cheapest plans ($k = 500$) for all robust plan selection strategies. For robust plan selection using the cardinality-slope metric FS and selectivity-slope metric SS, the robust plan candidates are further limited by the near-optimal plans ($\lambda = 1.2$).

Figure 4.14 shows some typical results for the Join Order Benchmark queries. The queries are plotted along the x-axis in the same order as in the Figures 4.10 and 4.12. The y-axis shows the $\rho_{c_{err,P}}$ in percent and $\delta_{c_{err,P}}$ in log-scale. The query set contains the queries with the best $\rho_{c_{err,P}}$ and $\delta_{c_{err,P}}$, e.g., Q14 and Q23, and the worst $\rho_{c_{err,P}}$ or $\delta_{c_{err,P}}$, e.g., Q16, Q23, Q25, and Q32. Overall, the robust plan selection with the cardinality-slope metric FS and selectivity-slope metric SS achieves a $\rho_{c_{err,P}}$ of 100% for 13 of the 31 executed queries, i.e., the robust plan selection chooses the most robust plan. A $\rho_{c_{err,P}} \geqslant 80\%$ is achieved for FS and SS in 25 of the 31 executed queries. In contrast, conventional plan selection choosing the estimated optimal plan EO achieves a $\rho_{c_{err,P}} \geqslant 80\%$ for only 12 of the 31 executed queries. The average $\delta_{c_{err,P}}$ over all 31 Join Order Benchmark queries is better for SS (−0.11) and FI (−0.12) compared to EO (−0.50). Considering the fastest plan FA, we again observe that it is not necessarily as robust as the estimated most robust plans. The average $\rho_{c_{err,P}}$ over all 31 Join Or-

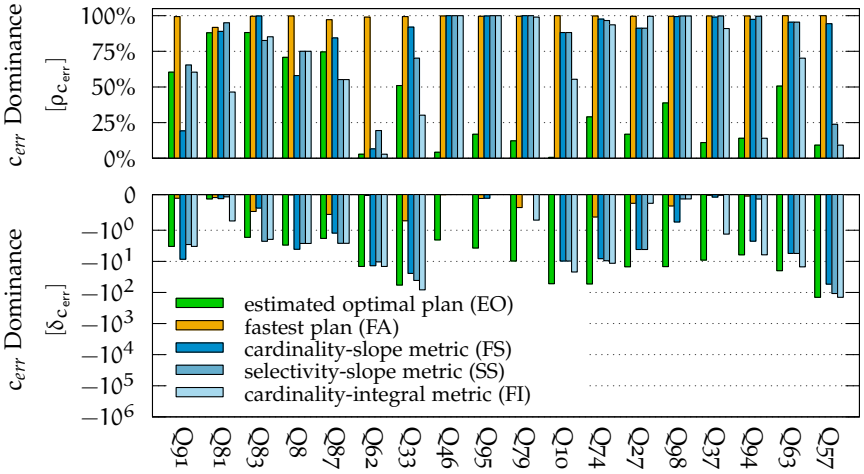


Figure 4.15: Typical results of experiment comparing the cost error factor dominance of the estimated optimal query execution plan (EO), the fastest query execution plan (FA), and the estimated most robust query execution plans according to one of the three robustness metrics (FS, SS, and FI) on synthetic benchmark queries with random query topology.

der Benchmark queries for the selectivity-slope metric SS (92.65%) is larger compared to the average $\rho_{c_{err,P}}$ of the fastest plans FA (89.37%).

Figure 4.15 plots some typical results of the synthetic benchmark queries with a random topology. The queries are the same, and in the same order as in the Figures 4.11 and 4.13. Included are Q46 and Q95 with the best $\rho_{c_{err,P}}$ and $\delta_{c_{err,P}}$, as well as Q57 and Q62 with the worst $\rho_{c_{err,P}}$ or $\delta_{c_{err,P}}$. For Q79, the cardinality-slope metric FS and selectivity-slope metric SS identify the most robust plan with $\rho_{c_{err,P}} = 100\%$ and $\delta_{c_{err,P}} = 0.0$, whereas EO is a volatile plan with $\rho_{c_{err,P}} = 12.20\%$ and $\delta_{c_{err,P}} = -9.74$. A $\rho_{c_{err,P}} \geqslant 80\%$ is achieved for FS in 75 of the 100 executed queries, and for SS in 68 of the 100 executed queries. In contrast, EO achieves a $\rho_{c_{err,P}} \geqslant 80\%$ for only 10 of the 100 executed queries. The average $\delta_{c_{err,P}}$ over all 100 benchmark queries is better for FS (-4.38), SS (-5.16), and FI (-8.18) compared to EO (-10.32).

Table 4.3 summarizes the results of all benchmarks, by best, worst, and average $\rho_{c_{err,P}}$, and best, worst, and average $\delta_{c_{err,P}}$ of all queries in the corresponding benchmark. The robustness metrics FS and SS achieve a significantly larger average $\rho_{c_{err,P}}$ (79%–93%) compared to estimated optimal plans EO (21%–69%) for all benchmarks. The worst $\delta_{c_{err,P}}$ for EO is in most benchmarks substantially smaller compared to the robust plans. A comparison of best, worst, and average $\rho_{c_{err,P}}$ and $\delta_{c_{err,P}}$ of FI, to FS and SS in Table 4.3 shows that FS and SS outperform FI. Again, the reason is that FI balances robustness and cost. However, best, worst, and average $\rho_{c_{err,P}}$ and $\delta_{c_{err,P}}$ of FI reveal a substantially better c_{err} dominance compared to estimated optimal plans EO.

		$\uparrow \rho_{c_{err}}$	$\downarrow \rho_{c_{err}}$	$\varnothing\rho_{c_{err}}$	$\uparrow \delta_{c_{err}}$	$\downarrow \delta_{c_{err}}$	$\varnothing\delta_{c_{err}}$
JOB	EO	100.00%	10.71%	68.92%	0.00	−2.16	−0.50
	FS	100.00%	28.00%	88.95%	0.00	−14.68	−0.52
	SS	100.00%	30.00%	92.65%	0.00	−1.53	−0.11
	FI	100.00%	10.71%	89.32%	0.00	−1.53	−0.12
	FA	100.00%	35.71%	89.37%	0.00	−0.40	−0.08
Chain	EO	91.60%	0.40%	31.75%	0.00	−20.69	−0.94
	FS	100.00%	48.00%	93.09%	0.00	−2.82	−0.11
	SS	100.00%	25.20%	89.37%	0.00	−4.49	−0.18
	FI	100.00%	6.40%	63.97%	0.00	−2.71	−0.38
	FA	100.00%	18.80%	92.95%	0.00	−1.47	−0.07
Cycle	EO	78.20%	0.20%	20.63%	−0.34	−26.27	−6.04
	FS	100.00%	44.30%	91.53%	0.00	−7.81	−1.02
	SS	100.00%	33.00%	90.50%	0.00	−6.05	−0.80
	FI	100.00%	0.20%	63.21%	0.00	−13.60	−2.71
	FA	100.00%	36.60%	95.13%	0.00	−5.42	−0.36
Star	EO	91.60%	0.20%	35.83%	0.00	−68.72	−2.01
	FS	100.00%	19.60%	78.84%	0.00	−13.61	−0.79
	SS	100.00%	3.40%	87.87%	0.00	−20.35	−0.66
	FI	100.00%	0.20%	63.79%	0.00	−20.35	−1.08
	FA	100.00%	13.60%	89.89%	0.00	−2.38	−0.20
Snowflake	EO	97.40%	0.20%	39.16%	−0.02	−41.33	−2.51
	FS	100.00%	36.20%	86.50%	0.00	−30.62	−1.22
	SS	100.00%	6.60%	88.87%	0.00	−40.72	−1.01
	FI	100.00%	1.80%	74.26%	0.00	−40.72	−1.74
	FA	100.00%	27.00%	93.91%	0.00	−3.73	−0.30
Random	EO	98.60%	0.60%	46.72%	−0.10	−143.02	−10.32
	FS	100.00%	6.60%	84.96%	0.00	−54.00	−4.38
	SS	100.00%	19.40%	83.71%	0.00	−108.07	−5.16
	FI	100.00%	2.80%	66.66%	0.00	−143.02	−8.18
	FA	100.00%	52.00%	97.46%	0.00	−13.61	−0.58

Table 4.3: Summary of cost error factor dominance comparisons between the estimated optimal plans (EO), the most robust plans according to cardinality-slope (FS), selectivity-slope (SS), or cardinality-integral metric (FI), and the fastest plans (FA) over all queries of different real-world and synthetic benchmarks.

4.6.2.3 *Correlated Cost Error Factor Limit*

According to the third consistency requirement, a large c_{err} for a plan with a small robustness value indicates a failure of the metric (see Section 4.2). Since cardinality estimations can be precise and always result in a small cost error factor c_{err}, even if a large robustness value is assigned, the correlation between the robustness value and c_{err} cannot be used to evaluate the third requirement. To evaluate the requirement, we draw all robust plan candidates of a query into a single plot. We do this analysis for the cardinality-slope and selectivity-slope robustness metrics, because they purely focus on plan robustness. The cardinality-integral metric takes cost as additional dimension into account, which does not allow a decoupled analysis. Corresponding to the previous experiments, the robust plan candidates are limited by the k-cheapest plans and near-optimal plans. Figure 4.16 shows the results for the cardinality-slope robustness metric FS, and Figure 4.17 for the selectivity-slope robustness metric SS. We show some typical results of Join Order Benchmark queries, namely Q7, Q8, Q12, Q14, Q19, and Q25. The robustness values r_{δ_f} and r_{δ_s} assigned by the robustness metrics are plotted on the x-axes in logarithmic scale, and the c_{err} that was calculated after plan execution on the y-axes in logarithmic scale. Additionally, we highlight the estimated optimal plan, the fastest plan and the estimated most robust plan according to the corresponding robustness metric. The Figures 4.16 and 4.17 also show the cost error factor improvement and cost error factor dominance, which correspond to the results in the Figures 4.12 and 4.14.

The results of the cardinality-slope robustness metric in Figure 4.16, reveal a strong correlation between r_{δ_f} and c_{err} for Q8, Q12, and Q14. Furthermore, there is no robust plan candidate with a small r_{δ_f} and a large c_{err}. The plots also confirm the results of Figure 4.14, where FS achieves a cost error factor dominance of 100%, because there are no plans with a smaller c_{err} than the estimated most robust plan. For Q7 and Q19, we see that the correlated cost error factor limit requirement is fulfilled, even if there is no strong correlation between r_{δ_f} and c_{err}. For Q25, the cardinality-slope robustness metric fails, which we also see in the previous experiments, i.e., the Figures 4.10, 4.12, and 4.14. The reason is that especially the $m{:}n$ joins in Q25 related to the `MOVIE_INFO` table create stronger cardinality estimation errors compared to other joins. The cardinality-slope robustness metric does not consider the joins related to the `MOVIE_INFO` table to create stronger estimation errors, and assigns a good robustness value to plans where the corresponding joins are deep in the plan. The result are strong estimation errors on deep plan edges that are propagated. Consequently, the plans have larger true costs $\mathring{c}$ and a worse c_{err}.

Figure 4.17 shows that the selectivity-slope robustness metric performs much better for Q25, because it assigns a good robustness value only to the plans that have a small c_{err}. The reason is that

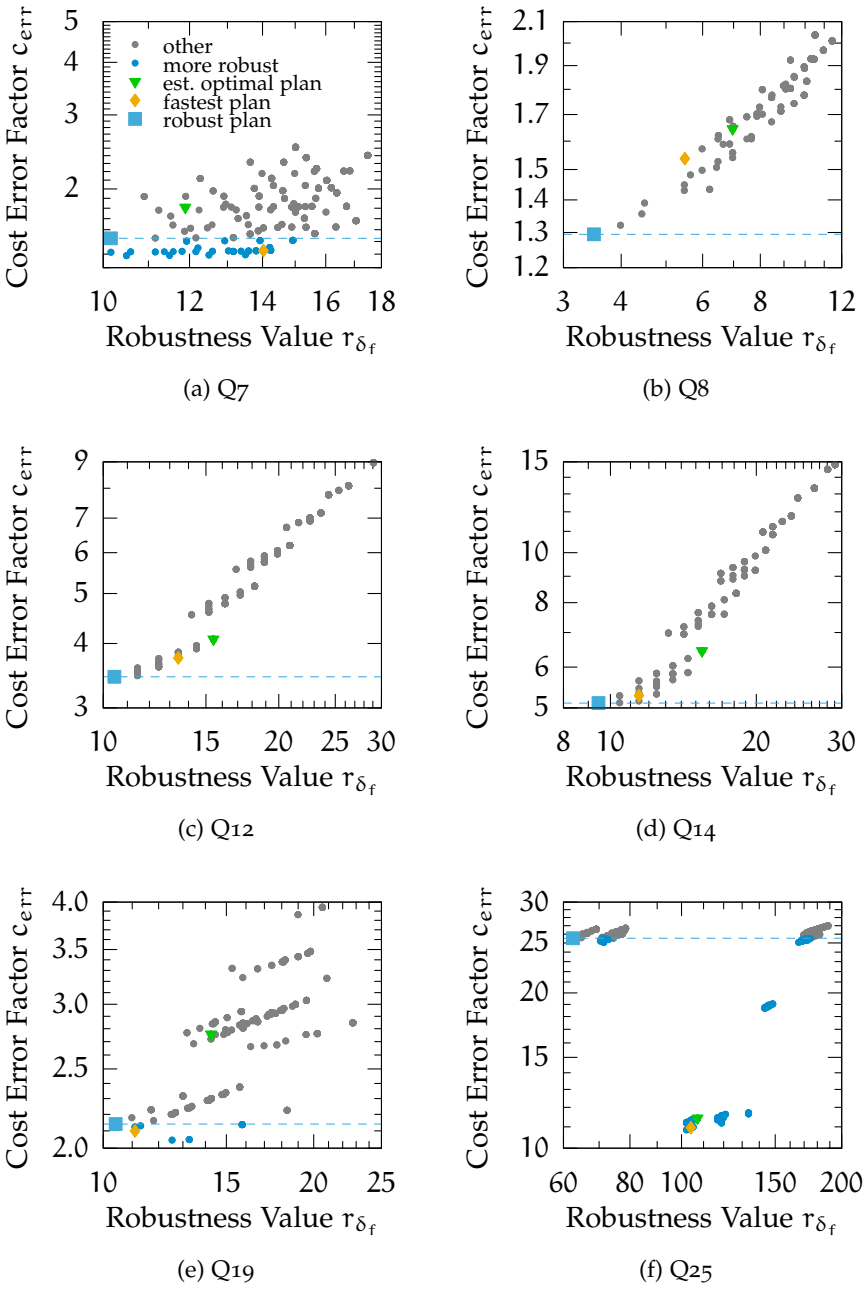
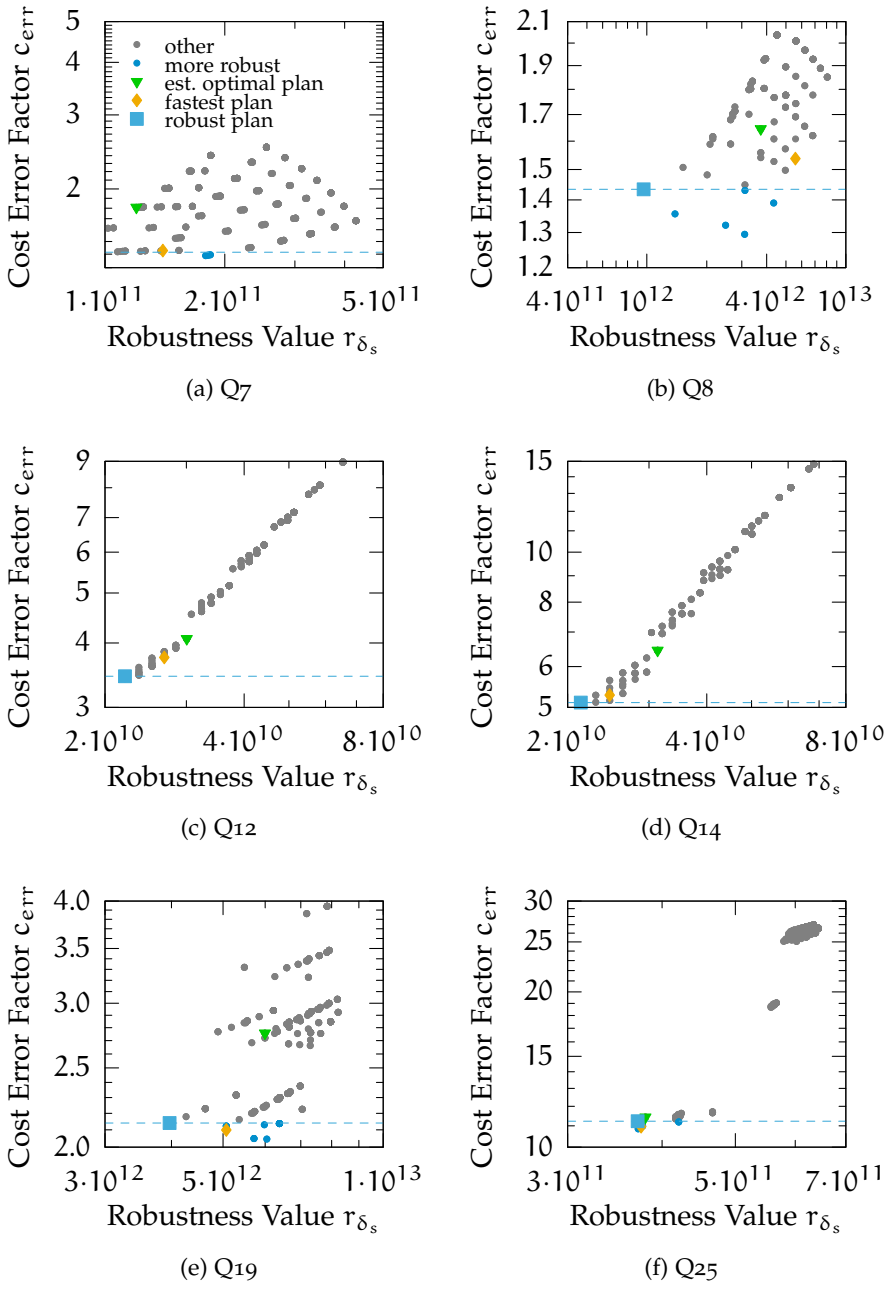


Figure 4.16: Experiment showing the correlated cost error factor limit of the cardinality-slope robustness metric for some typical results of the Join Order Benchmark.

the selectivity-slope robustness metric considers the risk of a large absolute cardinality error Δf on all edges. Consequently, it assigns a good robustness value only to the plans that perform the joins related to the `MOVIE_INFO` table on high plan edges. Since Figure 4.17f shows the same robust plan candidates as Figure 4.16f, the plans with strong cardinality estimation errors on deep plan edges now form a cluster. This is also a desirable behavior, because it reveals a strong correlation



(a) Q7 (b) Q8

(c) Q12 (d) Q14

(e) Q19 (f) Q25

Figure 4.17: Experiment showing the correlated cost error factor limit of the selectivity-slope robustness metric some typical results of the Join Order Benchmark.

between r_{δ_s} and c_{err}. The estimated most robust plan, the estimated optimal plan and the fastest plan have a small r_{δ_s} and result in a small c_{err}. A majority of other robust plan candidates have a large r_{δ_s} and result in a large c_{err}. Furthermore, there is no plan with a small r_{δ_s} that results in a large c_{err} for Q25. The results for Q7, Q12, Q14, and Q19 in Figure 4.17 are similar to results of the cardinality-slope robustness metric in Figure 4.16. The selectivity-slope robustness met-

ric performs slightly worse for Q8 compared to the cardinality-slope metric, but fulfills the correlated cost error factor limit requirement.

The plots for the other Join Order Benchmark queries look similar. In sum, the experiments on plan robustness reveal that our robustness metrics have the preferable behavior, which we describe in the consistency requirements in Section 4.2.

4.6.3 *Robust Plan Candidates*

In this section, we evaluate the impact of the robust plan candidate choice on the end-to-end query execution time and on the plan robustness. For the cardinality-slope FS, selectivity-slope SS, and cardinality-integral robustness metric FI, the robust plan candidates are the k-cheapest plans ($k = 500$). For FS and SS, the robust plan candidates are further limited to the near-optimal plans with $\lambda = 1.2$, because only FI considers cost. The query execution time experiments in Section 4.6.1 illustrate that robust plan selection with FS and SS suffers less from estimation errors compared to conventional plan selection EO, which chooses the estimated optimal or cheapest plan.

Our robust plan selection can be done at query optimization time, because each of our metrics has a low calculation effort, and we consider at most the k-cheapest plans as robust plan candidates. Table 4.4 shows the query optimization time relative to end-to-end query execution time, i.e., optimization time divided by end-to-end query execution time, for conventional plan selection EO and the robust plan selection using the FS, SS, or FI robustness metric. The query optimization time of conventional plan selection EO contains the plan enumeration, the cost calculations and the selection of the estimated cheapest plan. The query optimization time of robust plan selection contains the plan enumeration including the robust plan candidates, the cost calculations, the robustness value calculation for each robust plan candidate, and the selection of the estimated most robust plan. The optimization time depends on the number of enumerated plans, i.e., the query graph topology, and the number of robust plan candidates, which is at most $k = 500$ in all our experiments. We consider the Join Order Benchmark and the different topologies of the synthetic benchmark. The reported numbers are the average of all queries in the corresponding query set. Since robust plan selection introduces additional computational overhead, this ratio is better for conventional plan selection EO compared to robust plan selection using the FS, SS, or FI metric. The percentage of optimization time for star queries is smaller in comparison to all other query sets, which is counter-intuitive. The reason is that the total query execution time of the star queries is considerably larger, which we also illustrate in Table 4.1. However, the optimization time of robust plan selection is in any case small with respect to the end-to-end query execution time.

	JOB	Chain	Cycle	Star	Snowflake	Random
EO	0.98%	0.14%	0.12%	0.12%	0.17%	0.13%
FS	3.94%	3.70%	3.10%	1.77%	3.06%	3.08%
SS	4.78%	3.51%	3.32%	1.82%	3.20%	3.10%
FI	4.91%	3.68%	3.62%	1.71%	3.11%	2.96%

Table 4.4: Average query optimization time of conventional plan selection (EO) and robust plan selection using the cardinality-slope (FS), selectivity-slope (SS), or cardinality-integral robustness metric (FI), relative to the average end-to-end query execution time of Join Order Benchmark queries and synthetic benchmark queries.

Finally, we demonstrate that selecting the estimated most robust plan from the k-cheapest plans with $k = 500$ is competitive with respect to a robust plan that is selected form a larger set of candidates. Since setting $k = \infty$ is infeasible, especially for the complex query graph topologies of some Join Order Benchmark queries, we limit k for this experiment to $10,000$. We denote the difference between the cost error factor c_{err} of the estimated most robust plan with $k = 10,000$ and the estimated most robust plan with $k = 500$ as $\gamma_{c_{err}}$. A negative $\gamma_{c_{err}}$ indicates that robust plan selection found a more robust plan with a larger k. A $\gamma_{c_{err}}$ close to 0 indicates that robust plan selection found no considerably more robust plan with a larger k.

Table 4.5 shows the average $\gamma_{c_{err}}$ for robust plan selection with our three robustness metrics for both, the Join Order Benchmark and the synthetic benchmark. In the Join Order Benchmark, the average $\gamma_{c_{err}}$ is close to 0 for the selectivity-slope SS and cardinality-integral robustness metric FI, i.e., a larger k will not result in substantially more robust plans. For the cardinality-slope robustness metric FS, the average $\gamma_{c_{err}}$ is even positive. The reason is that robust plan selection with a larger k will choose a plan for Q15 and Q21 that results in a significantly larger c_{err}. In the synthetic benchmark, the average $\gamma_{c_{err}}$ is close to 0 for all three robustness metrics on chain and cycle queries. For star and snowflake queries, the average $\gamma_{c_{err}}$ value is smaller compared to chain and cycle queries due to the larger plan space. Finally, for random queries, the average $\gamma_{c_{err}}$ is close to 0 for the cardinality-integral robustness metric FI. In contrast, robust plan selection with the cardinality-slope robustness metric FS will lead to a $\gamma_{c_{err}}$ smaller than -1 for 43 of the 100 generated queries, and with the selectivity-slope robustness metric SS for 57 of the 100 generated queries. Overall, $k = 500$ achieves a good trade-off between plan robustness and query execution time, because the $\gamma_{c_{err}}$ is close to 0 for a large number of queries, and the optimization overhead is small. Especially for complex queries, $k = 10,000$ can cause a significantly larger optimization time with respect to the sheer plan execution time.

	JOB	Chain	Cycle	Star	Snowflake	Random
FS	+5.86	−0.05	−0.39	−1.39	−1.91	−3.04
SS	−0.85	−0.08	−0.18	−0.71	−1.75	−2.62
FI	−1.00	−0.04	0.00	−0.33	−0.48	−0.24

Table 4.5: Average $\gamma_{c_{err}}$ of robust plan selection using the cardinality-slope (FS), selectivity-slope (SS), or cardinality-integral robustness metric (FI) and 500 robust plan candidates, relative to robust plan selection with 10,000 robust plan candidates for Join Order Benchmark queries and synthetic benchmark queries.

For Q25 of the Join Order Benchmark, the optimization takes a factor of 60 longer compared to the execution, when $k = 10,000$.

From our experiments we conclude that the selectivity-slope robustness metric has advantages over the cardinality-slope robustness metric. There are queries for which the one or the other metric has stronger speedups or stronger regressions, but the example of Join Order Benchmark Q25 shows that the additional features of the selectivity-slope robustness metric can prevent significant outliers. Although there are some queries for which the cardinality-integral robustness metric achieves the best results, the overall performance of the cardinality-slope and selectivity-slope robustness metrics is better. The advantage of the cardinality-integral metric is the support for arbitrary PCF shapes, and it is independent of the estimated cardinality $\hat{f}$. In sum, we recommend the selectivity-slope robustness metric for linear PCFs, and the cardinality-integral metric for non-linear PCFs.

4.7 RELATED WORK

Open research questions in robust query processing are regularly discussed in *Dagstuhl seminars* organized by Graefe and colleagues [60, 65, 91]. As classified by Yin et al. [80], one approach to robust query processing is robust plan selection. The design space for robust plan selection strategies has similarities to the design space of conventional query optimizers. We argue that the design space of robust plan selection strategies has the following three dimensions, in which related work can be categorized:

1. *Online Selection* vs. *Offline Analysis:* A robust plan can be either selected at optimization time, i.e., online selection, or identified in a more expensive offline analysis.

2. *Robust Plan Candidates:* The set of candidates for robust plans can be limited, e.g., to plans that are only optimal for certain cardinalities and selectivities [53, 82], plans that have costs close to the estimated optimal plan [59], or plans with a certain query execution plan structure, e.g., only left-deep trees [77, 81].

3. *Selection of the Robust Plan:* There are numerous approaches to select the most robust plan from the set of robust plan candidates. Selected could be a plan that is optimal for multiple cardinality and selectivity combinations [41], or the most robust plan according to a robustness metric [59, 77, 81].

4.7.1 *Offline Analysis*

Robust Plan Diagram Reduction [53] and *Plan Bouquets* [71] reduce parametric optimal sets of plans (POSP) [37]. Robust Plan Diagram Reduction is a graphical plan space analysis that identifies robust plan clusters. Plan Bouquets iteratively explore different plans through execution, give a formal upper bound for execution time compared to the fastest plan, and do not rely on cardinality estimation. Enumerating POSPs and identifying the Plan Diagram or the Plan Bouquets causes a very high pre-calculation effort, so that it is not reasonable for ad-hoc queries. Updates can require considerable effort. Our robust plan selection approach is based on estimations, and enables the specification of an upper bound for cost with respect to the estimated optimal plan through the near-optimal plans constraint. Due to the small pre-calculation effort, it can be applied at optimization time, and supports updates and ad-hoc queries.

Risk Score [82] is a metric for plan robustness, and indicates how fragile a plan during different execution conditions is. Since different execution times are necessary, a Risk Score cannot be predicted during optimization time. The robust plan candidates set is also limited to the POSP, i.e., plans that have to be optimal for some cardinalities.

4.7.2 *Online Selection*

Proactive Re-Optimization [41] searches the optimal plan for the estimated and two heuristically chosen cardinalities for each cardinality estimate. From the three plans, it tries to identify the optimal, a robust, or a switchable plan. If no such plan exists, it triggers a runtime re-optimization. The two heuristically chosen cardinalities for each cardinality estimate, are supposed to be an optimality range, which we can be precisely calculated with our algorithm in Chapter 3.

Robust Cardinality Estimation [40] searches the optimal plan for worst case cardinality estimates. It uses random sampling to generate a probability density function for operator output cardinality. This approach is limited to operators that support sampling. Based on the probability density function and a user defined risk level for the probability density function, it estimates the maximum output cardinality of an operator, and searches the optimal plan for it.

Compared to Proactive Re-Optimization and Robust Cardinality Estimation, our approach defines a robustness value so that two plans

can be compared with respect to their robustness. We also consider non-optimal plans in the robust plan candidates set, since a robust plan does not require optimality for certain cardinalities. Furthermore, the goal of our approach is robust cost behavior in case of over- and under-estimations, i.e., a minimal c_{err}. Robust Cardinality Estimation does not necessarily select a plan with minimal c_{err}.

Minmax Regret Rule [81] is similar to Proactive Re-Optimization, but considers more plans and has a different robust plan selection criterion. It compares the costs of the plans at different cardinalities. Selected is the plan that has the smallest maximum cost difference to the optimal plans, over all cardinalities. Due to the increased number of plans, it is limited to left-deep trees. Since this limitation excludes possible robust plans, we consider all plan trees in our work.

An extension to the Minmax Regret Rule are *Cost-Stable Plans* [59], which choose the plan with the smallest average cost difference to the estimated optimal plans, over all cardinalities. In addition, Cost-Stable Plans limit the number of plans, e.g., by early pruning of outliers that have a large cost difference to the estimated optimal plan.

Due to its efficiency, our robust plan selection approach can be applied at optimization time, i.e., it does online selection. Compared to other approaches, we are not limited to certain tree structures [81] or plans that are optimal for some cardinalities [40, 41, 53, 82]. We limit the number of robust plan candidates to the cheapest plans encountered during the initial query optimization. In contrast to competing approaches [59, 81], we can assign robustness values independent of other query execution plans. Finally, we define robustness metrics that work with classical single point estimation and are not bound to more expensive cardinality estimation techniques [40, 41, 59, 81].

4.8 CONCLUSION

The three novel robustness metrics we present in this chapter are valuable building blocks for robust query processing. They efficiently quantify the robustness of query execution plans at optimization time and consider the impact of potential cardinality estimation errors during the query execution plan selection. Compared to competitive approaches for robust plan selection, we do not limit the plan tree structure, can calculate a robustness value for a single plan independent of other plans, and are not bound to expensive statistical models. While optimality ranges, for which we present a calculation algorithm in Chapter 3, can only evaluate the robustness of optimal query execution plans, the robustness metrics support non-optimal plans.

Despite the simplicity of all three robustness metrics, our experimental evaluation demonstrates their effectiveness. Furthermore, we observe that robust plans can outperform the estimated cheapest plans in the presence of cardinality estimation errors. In our experiments,

the robustness plans can speedup queries on a real-world database by up to 1.83×, and queries of a synthetic benchmark by up to 3.49×. In scenarios with few cardinality estimation errors, the near optimal plans requirement guarantees at most minor regressions of estimated costs. Our experiments furthermore reveal that the overhead to enumerate the robust plan candidates and calculate the robustness values consumes on average a single-digit percentage of the end-to-end query execution time. At the same time, we can improve the cost error factor by up to 90, while its degradation is at most 25. We further demonstrate that all robustness metrics can identify robust plans, which dominate all robust plan candidates with respect to their cost error factor. Depending on the benchmark, the most robust plan according to the cardinality-slope or selectivity-slope robustness metric reaches an average cost error factor dominance between 79% and 93%. Finally, our formal problem specification and requirements build a solid foundation for future research on robust metrics.

5 MID-QUERY RE-OPTIMIZATION REVISITED

The robustness metrics and robust plan selection in Chapter 4 work with estimated cardinalities. In this chapter, we use true cardinalities, collected at query execution time, to compensate sub-optimal query execution plans, caused by cardinality estimation errors. We demonstrate that true cardinalities are superior to estimations and can further improve the query execution plan quality.

5.1 INTRODUCTION

In this chapter we revisit Mid-Query Re-Optimization [31, 39, 41], an Adaptive Query Processing technique [47, 80], which collects true cardinalities of intermediate results at query execution time. It uses the true cardinalities to re-optimize the query execution plan and compensate cardinality estimation errors. Like a regular query processor, a Mid-Query Re-Optimizer starts executing the estimated cheapest, i.e., optimal plan, selected by the query optimizer. At certain points in the plan, it collects the true cardinalities of intermediate results, and invokes the query optimizer again with the true cardinalities to search for a better query execution plan. If a better plan is identified, the query execution engine switches to the better plan.

Like the majority of Adaptive Query Processing approaches [80], Mid-Query optimization requires modifications to the query processor, to collect cardinalities in the execution engine, insert cardinalities back into the optimizer, and switch plans at query execution time. Compared to other adaptive query processing approaches that need to monitor multiple query executions [36], Mid-Query Re-Optimization has the advantage that it can already improve the first execution of a query. Consequently, it is applicable to ad-hoc queries.

Our goal is to improve the query execution plan quality, and thus execution time, by exploiting the knowledge of true cardinalities. We formalize this problem in Section 5.2. There are multiple aspects that introduce a risk of end-to-end query execution time degradations. Re-optimizations and plan switches can consume additional execution time. Furthermore, the calculation of redundant intermediate results, or modifications on the highly-optimized pipeline execution can cause a degradation of end-to-end query execution time. Our objective is to keep the risk of query execution time degradations low. The baseline in our experiments is the estimated optimal plan, executed as fast as possible in a conventional query processor. In case the estimated optimal plan is the true optimal plan, we only want to intro-

duce a few additional optimizer calls, but no considerable overheads. We identify three design choices within Mid-Query Re-Optimization, that impact the risk of query execution time degradations:

RE-OPTIMIZATION POINT The first design choice is on the point in the query execution plan where re-optimizations and plan switches take place. Like Kabra and DeWitt [31], we only re-optimize and switch plans at the end of pipelines, i.e., in the pipeline breakers. As we further discuss in Section 5.4.1, in pipeline breakers the intermediate results are completely materialized and all parallel tasks to execute the pipeline are finished. Unlike related approaches, we never introduce additional, artificial pipeline breakers [39, 69] because they can increase the query execution time.

EAGER RE-OPTIMIZATION Complementary to the re-optimization point is the choice whether an intermediate result has to be calculated completely, before a re-optimization is triggered. To derive the true cardinality, an intermediate result has to be complete. Re-optimizing on incomplete intermediate results may cause wrong optimization choices. Further, the partially calculated intermediate results may be discarded. Unlike Avnur et al. [32] and Markl et al. [39], we re-optimize only on complete intermediate results and true cardinalities.

RE-OPTIMIZATION SCOPE The second design choice is whether the entire query is re-optimized or only those query parts that have not yet been executed. Like Neumann and Galindo-Legaria [69], we do the latter. We further explain in Section 5.4.1 that re-optimizing the entire query introduces the risk of wasting already calculated intermediate results, and therefore query execution time. Consequently, we limit the re-optimizations on the not-yet-executed query parts, and always reuse the already calculated intermediate results.

Related approaches on Mid-Query Re-Optimization [31, 39, 49, 69] were integrated into existing query processors, in which query optimizer and query execution engine are strictly separated components. The only connection is the query execution plan that is passed from the optimizer to the execution engine for execution. Consequently, re-optimizations and plan switches are costly: The query execution engine aborts the current query execution and calculated intermediate results are stored as materialized views. The optimizer is invoked for the entire query, taking the materialized views into account to search a new optimal query execution plan. This re-optimization enumerates also plan classes that are not affected by the calculated intermediate results. The new plan is passed to the execution engine to be scheduled for execution. Since this procedure is expensive, related work on Mid-Query Re-Optimization focuses on the definition of heuristics

for optimality ranges, to decide when to trigger a re-optimization. We observe a trade-off between the magnitude of modifications on the query processor, and the execution time overhead of adaptions. We argue that a better integration of query optimizer and query execution engine implies smaller execution time overheads of adaptions.

Our major contribution is an adaptive query processor, which increases the efficiency of re-optimizations and plan switches, through an improved interaction between query optimizer and query execution engine. We argue that more efficient re-optimizations and query execution plan switches enable a larger number of re-optimizations, and consequently a larger number of query execution plan switches, which ideally improve the query execution plan quality and query execution time. Like in the Chapters 3 and 4, we demonstrate our concepts for join operators. Nevertheless, all concepts we present in this chapter are applicable to other relational operators.

We provide a formal problem description in Section 5.2, and further contributions form the building blocks of our adaptive query processor. The initial building block is the adaptive execution strategy, which we explain in Section 5.4.1. In Section 5.4.2, we explain the extension of the CHT hash join [70] (see Section 2.5.1.1), which makes it NUMA-aware and enables plan switches within the build phase of the hash table. In Section 5.4.3 we explain the selective re-enumeration algorithm, which is an extension of the DPsize enumeration algorithm. It gives the execution engine an interface to inject runtime cardinalities into the dynamic programming table and trigger a re-enumeration of only the plan classes that are affected by the cardinality update. Section 5.4.4 eventually explains the re-optimization criteria. Another contribution next to our adaptive query processor in Section 5.4 is our experimental evaluation in Section 5.5, which shows the speedup of query execution that the query processor achieves. After discussing related work in Section 5.6, we conclude our findings in Section 5.7.

5.2 FORMAL PROBLEM DESCRIPTION

We formalize the problem of improving the not-yet-executed query parts, at query execution time, with true cardinalities. We denote the procedure to decide whether there is a re-optimization, the re-optimization itself, and the potential plan switch as potential adaption A_i, where $i \in \mathbb{N}^+$. Furthermore, we denote the initial query execution plan as P_0, and the query execution plans after a potential adaption A_i as P_i. The true costs of plans P_i are denoted as $\mathring{c}_i$. A potential adaption A_i can cause a plan switch so that $P_i \neq P_{i-1}$. An adaptive query processor starts executing the initially estimated opti-

mal plan P_0. In case $P_i \neq P_{i-1}$, the adaptive query processor should only switch from Plan P_{i-1} to P_i, if:

$$\mathring{c}_i \leqslant \mathring{c}_{i-1}. \tag{5.1}$$

Therefore a plan switch should never increase the true cost. This criteria can be checked after both plans P_i and P_{i-1} are executed and their true cost are calculated. Since all adaptive query execution plans are based on an initial query execution plan, we denote the initial optimization time of all plans P_i as t_{opt}. We next assume that query execution plans P_i can be executed entirely in the query execution engine. We denote the execution time, which the query execution engine needs to execute the entire plan P_i as $t_{i,exec}$. It consists of the time to execute all pipelines in the plan and negligible time to make orchestration decisions. Furthermore, we denote the execution time of a potential adaption A_j as $t_{j,adapt}$. Since we assume to re-optimize only the not-yet-executed query parts, the end-to-end query execution time of plan P_i can be defined as the sum of initial optimization time t_{opt}, the execution time of an entire plan P_i in the execution engine $t_{i,exec}$, and the time for all potential adaptions A_j:

$$t_i = t_{opt} + t_{i,exec} + \sum_{j=1}^{i} t_{j,adapt}. \tag{5.2}$$

Assuming an accurate cost model, an improvement in true cost should always result in a faster execution time $t_{i,exec}$:

$$\mathring{c}_i < \mathring{c}_{i-1} \implies t_{i,exec} < t_{i-1,exec}. \tag{5.3}$$

Furthermore, the effort of potential adaptions in an ideal adaptive query processor should be small, so that:

$$\mathring{c}_i < \mathring{c}_{i-1} \implies t_i < t_{i-1} \tag{5.4}$$

holds in practice. In case the adaptive query processor cannot identify a plan better than P_0, i.e., there are no plan switches and $P_i = P_0$, the end-to-end query execution execution time t_i of P_i is:

$$t_i = t_0 + \sum_{j=1}^{i} t_{j,adapt}. \tag{5.5}$$

Consequently, the adaption effort should be small. We nevertheless argue that missing better plans wastes more query execution time compared to re-optimizations that cannot identify a better plan. Consequently, an adaptive query processor should only avoid a re-optimization when it can guarantee that there are no better plans.

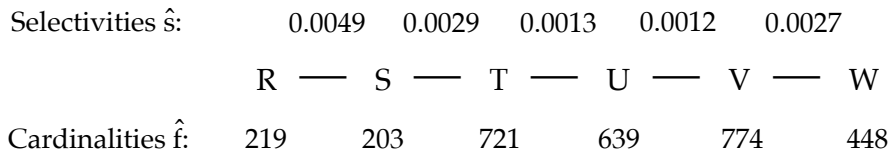


Figure 5.1: Motivating example: Chain query from the synthetic benchmark joining six relations with estimated cardinalities and selectivities.

5.3 ADAPTIVE PLAN EXAMPLE

To illustrate the improvements of true cost and query execution time, we present a motivating example of an adaptive query execution plan identified by our adaptive query processor. We consider a query of the synthetic benchmark we used to evaluate our calculation algorithm for optimality ranges in Section 3.3.2. It is a chain query joining six relations with estimated cardinalities and selectivities as shown in Figure 5.1. Figure 5.2 shows the initial estimated optimal plan together with the estimated cardinalities $\hat{f}$ and true cardinalities $\mathring{f}$ on all edges. We explain how we estimate the cardinalities of intermediate results in Section 2.6.3, and demonstrate the competitiveness our our cardinality estimator in Section 2.7. In Section 2.7, we furthermore explain why our query optimizer's choice of the estimated optimal plan is comparable to the choice in popular free and commercial database systems. Although our cardinality estimator is competitive, the query contains some expanding joins, which are significantly underestimated. The cardinality of $RS \bowtie T$ is underestimated by two orders of magnitude. This estimation error is propagated to the next join $RST \bowtie U$. After that, the join $RSTU \bowtie V$ is again expanding and the cardinality significantly underestimated, so that the estimation error reaches five orders of magnitude. In the end, the output cardinality is $30,424,722$ compared to the estimation of 484.

Remember that throughout this work, the build side of hash joins is the input on the right-hand side. Particularly bad about the query execution plan in Figure 5.2 is, that two build sides are on intermediate results with millions of tuples. In contrast, the corresponding probe sides have only a few hundred tuples. This is an example of a query execution plan that is reasonable for the estimated cardinalities, but that shows a very unfortunate behavior in case of cardinality estimation errors that are propagated through the plan.

Figure 5.3 shows the query execution plan that our adaptive query processor eventually executes. Initially, it is the same query execution plan as in Figure 5.2, in which the join between RS and T is significantly underestimated. The join result $RS \bowtie T$ is materialized in the build side of the next join in Figure 5.2, i.e., $RST \bowtie U$. At this point, our adaptive query processor invokes the optimizer again to search for a better plan to join RST with U, V, and W. The result is the bushy

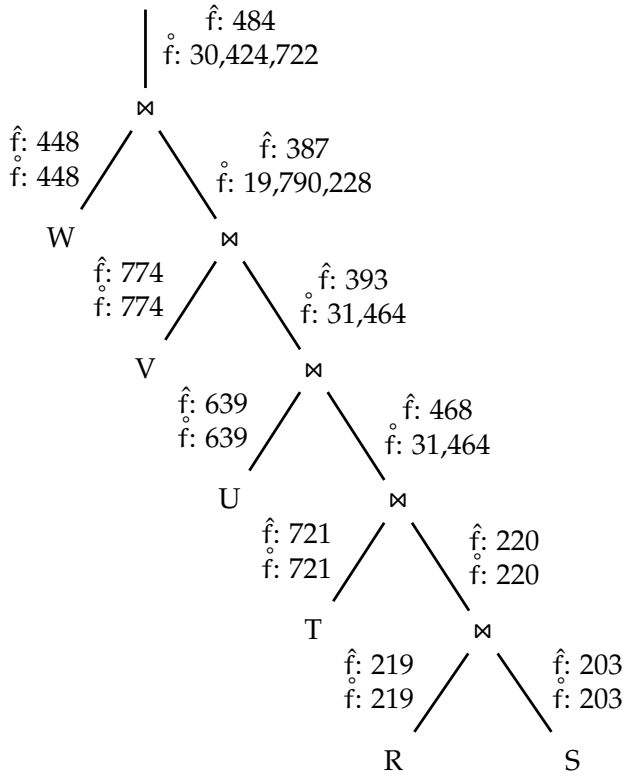


Figure 5.2: Initial estimated optimal query execution plan for motivating example query, showing estimated and true cardinalities.

query execution plan in Figure 5.3. It does not only have a different operator order, but also adjusted build and probe sides. Since RST is larger compared to the result of joining U, V, and W, it is now the probe side. Although both plans create the same result, comparing the true costs and end-to-end query execution times shows that the adaptive plan in Figure 5.3 is significantly cheaper and faster. For the C_{mm} cost function [93], the final adaptive plan has true cost of 30,463,486, and a query execution time of 0.16s, In contrast, the static plan in Figure 5.2 has true cost of 70,134,681 and finishes in 10.2s. According to the Equations 5.2 and 5.4 in our formal problem description, the improvements of true cost and end-to-end query execution time reveal a preferable behavior of an adaptive query processor.

5.4 ADAPTIVE QUERY PROCESSOR

The foundation of our adaptive query processor is the query processor that we explain in Section 2.7. While we use this query processor in Chapter 3 and 4 with the extended hash join operator, we further extend some central components, such as the plan enumerator, and the plan executor, to enable efficient re-optimizations and plan switches. We explain the building blocks top down, starting with the

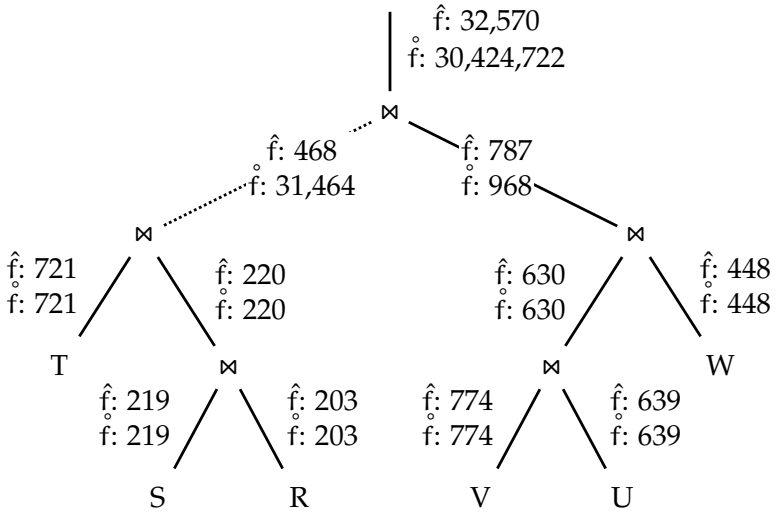


Figure 5.3: At query execution time after two joins re-optimized and adjusted query execution plan for motivating example.

adaptive execution strategy, followed by the extension of the CHT hash join, and finally the selective plan enumeration.

5.4.1 *Adaptive Execution Strategy*

In Section 5.1, we explain that the first design choice in Mid-Query Re-Optimization is about the re-optimization points in the query execution plan, and that we decide to only re-optimize at existing pipeline breakers. It is also possible to re-optimize after each operator. In a pipelined query execution, this results in additional, artificial pipeline breakers. Kabra and DeWitt [31], and Neumann and Galindo-Legaria [69] argue that artificial pipeline breakers can introduce considerable execution time overheads, but do not reveal numbers. We experimented with artificial pipeline breakers, and observed that arbitrarily large performance degradations can be constructed, depending on the size of the query and the number of artificial pipeline breakers. Artificial pipeline breakers interrupt the parallel pipeline execution and force the materialization of intermediate results, causing a loss of data cache locality. Any additional effort like this has to be compensated with a considerable query execution plan improvement, which cannot always be guaranteed. Consequently, re-optimizing anywhere else than at existing pipeline breakers significantly increases the risk of wasting query execution time. Existing pipeline breakers are perfect to do a re-optimization, because the pipelines can be executed as fast as possible, exploiting all optimizations we discuss in Section 2.5.

The second design choice in Mid-Query Re-Optimization is about eager re-optimizations. Eager re-optimizations are triggered on incomplete intermediate results, to avoid sub-optimal query execution

plans earlier. Since the derived cardinalities of incomplete intermediate results are not precise, eager re-optimizations can cause wrong optimization choices. Furthermore, the incomplete intermediate results might be discarded after a plan switch. In a pipelined query execution, eager re-optimizations can also result in synchronizations within the pipeline, and therefore performance degradations. Unlike Avnur et al. [32] and Markl et al. [39], we re-optimize only on complete intermediate results that are materialized in pipeline breakers. Re-optimizing on complete intermediate results in a pipeline breaker has the advantage that all parallel tasks are joined. Furthermore, the pipelines can be executed as fast as possible, and we avoid the redundant calculation of incomplete intermediate results. Another advantage is that the true cardinalities can be derived from the materialized intermediate results. Deriving the cardinality of an intermediate result, which is materialized in a pipeline breaker, is almost for free and does not add overheads in our adaptive query processor.

The third design choice in Mid-Query Re-Optimization is the re-optimization scope, i.e., the parts of the query that are re-optimized. We explain in Section 5.1 that we only re-optimize the not-yet-executed query parts, like Neumann and Galindo-Legaria [69] do. We argue that this re-optimization scope guarantees progress, and avoids the calculation of redundant intermediate results. Consequently, there is a small risk of execution time degradations with respect to the initial estimated optimal plan. Alternatively, the re-optimization scope could span across the entire query, which could be beneficial if the initial plan is far from the true optimum, because already the cardinalities of the first operators are considerably mis-estimated. In this case, searching for another plan, which could be completely different and does not consider the already calculated intermediate result, could be overall faster compared to continuing the initial plan. Nevertheless, there is no guarantee that such a plan is found. Searching a completely new plan can result in the calculation of multiple redundant intermediate results, before the final plan is identified. Although the redundant intermediate results can improve the re-optimizations, they can also increase the query execution time. We demonstrate in our experimental evaluation in Section 5.5 that re-optimizing only the not-yet-executed query parts creates stronger query execution time improvements than degradations, and also results in a significantly faster average query execution time. Based on these considerations, we present the following adaptive execution strategy.

Figure 5.4 illustrates the execution strategy of our adaptive execution engine. Each plan is executed pipeline by pipeline, and the main loop in Figure 5.4 orchestrates the execution of pipelines, the re-optimizations, and the plan switches. Figure 5.5 shows an example query execution plan, containing hash join operators with the build and probe sides as illustrated. It is the optimal execution plan of the

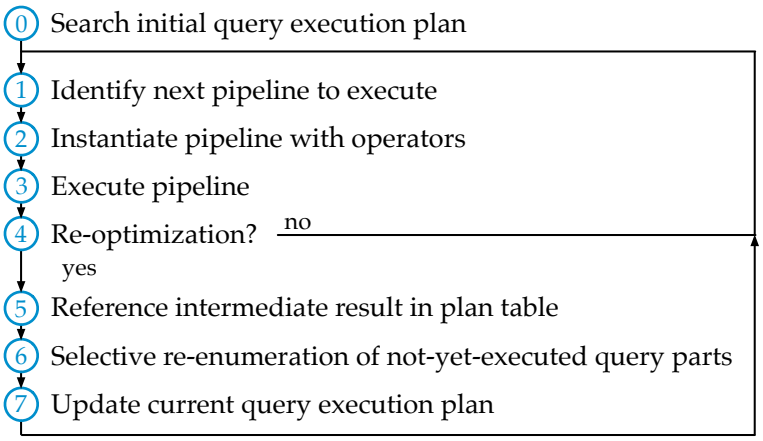


Figure 5.4: Execution strategy of our adaptive execution engine, orchestrating pipeline executions, re-optimizations, and plan switches.

query in Figure 3.1, which we use as running example to illustrate the calculation algorithm for optimality ranges. It is also the running example in this chapter. All query execution plans in our query processor consist of physical operators, such as hash join operators or base table scans. Each physical operator references its children, so that a query execution plan is just a reference to the top operator. Next, we explain the different steps of our adaptive execution strategy illustrated in Figure 5.4:

STEP 0 – INITIAL OPTIMIZATION Initially, the query execution engine invokes our query optimizer to search the optimal query execution plan based on estimated cardinalities. According to Section 2.7, we exhaustively enumerate bushy plans using our DPsize enumerator, with the C_{mm} cost function, and a basic cardinality estimator. The adaptive execution engine keeps the plan table of the query optimizer after the initial plan search to speed-up the re-optimizations.

STEP 1 – PIPELINE SELECTION The first step in the loop in Figure 5.4 is the identification of the next pipeline that is going to be executed. To keep the opportunities for re-optimizations high, we always execute only one pipeline of a query execution plan at a time. We explain in Section 2.7.2 that our query execution engine has a high level of parallelism within a single pipeline. Therefore, executing just one pipeline at a time is no considerable disadvantage. Since it is possible to execute multiple pipelines at a time, we have to choose in Step 1 which pipeline is executed next. In the example in Figure 5.5, there is one pipeline that is already executed, namely the pipeline that starts at the base table R and ends in the build side of $R \bowtie S$. Since a pipeline can only be executed as soon as all dependent pipelines are executed, the only candidates for the next execution are the pipelines starting

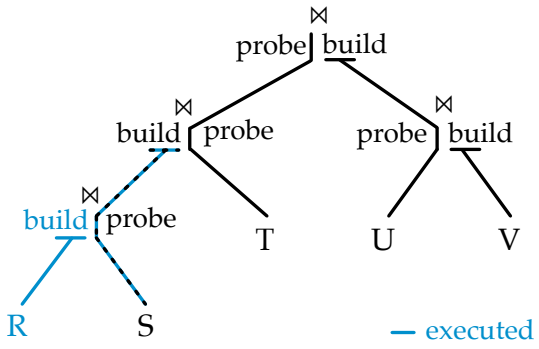


Figure 5.5: Optimal query execution plan of Query in Figure 3.1 as running example to illustrate the adaptive execution strategy.

on the base tables S and V. In our running example in Figure 5.5, the pipeline starting on base table S is selected.

We experimented with multiple pipeline selection strategies. We observed that a trivial traversal of the query execution plan tree can cause a sub-optimal memory utilization, because pipelines could be executed too early so that the pipeline breaker occupies memory longer than actually necessary. In the example in Figure 5.5, executing the pipeline of V and then the pipeline of S would be inefficient. The breaker at the end of the V pipeline containing an intermediate result would occupy memory for the entire execution time of the S pipeline, although the V pipeline can also be executed after the S pipeline. Consequently, we use a more memory-efficient pipeline selection strategy, which avoids that pipelines are executed before they are actually needed. For each new plan, we create a dependency graph of pipelines, in which each edge is weighted with the number of directly or indirectly required pipelines. To identify the next pipeline for execution, we perform a depth-first search for a not-yet-executed pipeline in the dependency graph. During the depth-first search we first traverse the edges, which have the highest weight, i.e., the largest number of directly or indirectly required pipelines.

Another option, similar to the work of Neumann and Galindo-Legaria [69], is to prefer the pipeline, whose intermediate result cardinality has the highest impact on the plan choice. Since Neumann and Galindo-Legaria [69] restrict the join trees to linear trees, we prefer the memory-efficient pipeline selection strategy.

STEP 2 – PIPELINE INSTANTIATION Step 2 in the loop in Figure 5.4 is the instantiation of the previously selected pipeline. The instantiation contains the creation of a pipeline object, to which all operators and the pipeline breaker are added. Our adaptive execution strategy also supports other kinds of pipelines, such as just-in-time compiled pipelines that we explain in Section 2.5.2.

STEP 3 – PIPELINE EXECUTION The third step in the loop is the parallel execution of the pipeline, as we describe it in Section 2.7.2. Each pipeline is executed without being affected by adaptivity features. After the pipeline is executed, the pipeline breaker contains a complete intermediate result and its true cardinality, so that re-optimizations and plan switches are reasonable. We explain in Section 5.4.2 that each pipeline breaker is based on a generic pipeline breaker which just collects all the batches that are pushed to it. Since we maintain the number of valid tuples in each batch, it is comparably cheap to determine the true cardinality of an intermediate result. We only determine the true cardinality because cardinality has the strongest impact on the cost of query execution plans.

It is of course possible to collect further runtime statistics, such as the true distinct count. The true distinct count could be beneficial to update the selectivity estimates of joins that are related to the intermediate result, and therefore the cardinality estimates of the following intermediate results. Nevertheless, deriving the true distinct count is considerably more expensive compared to the true cardinality. In contrast to cardinality, the distinct count can be different for different columns, which adds another dimension of overhead, even if only the interesting columns are considered. It remains open to analyze whether the additional effort to collect runtime statistics other than cardinality is reasonable, and whether it might be sufficient to derive only improved estimates, such as sketches [64] for distinct count.

STEP 4 – RE-OPTIMIZATION CRITERIA Step 4 in the loop decides whether there is a re-optimization. We illustrate the corresponding re-optimization criteria in Section 5.4.4. In case there is no re-optimization, we continue with the next loop iteration, i.e., identify, instantiate, and execute the next pipeline. In case there is a re-optimization, we continue with next step in the loop.

STEP 5 – INTERMEDIATE RESULT REFERENCING The fifth step in the loop starts with wrapping the intermediate result into a special physical operator, denoted as *intermediate result operator*. The intermediate result operator is referenced as optimal plan in the corresponding plan class in the plan table, which the query execution engine kept from the initial plan search. At the same time, the estimated cardinality of the plan class is replaced by its true cardinality, which is derived from the calculated intermediate result. Referencing the intermediate result operator as optimal plan in the corresponding plan class guarantees that the intermediate result is not lost, and that its true cardinality is considered in the next step.

STEP 6 – SELECTIVE RE-ENUMERATION Step 6 in the loop is the selective re-enumeration, which we illustrate in detail in Section 5.4.3.

It enumerates only those plans which are affected by the updated cardinality of the intermediate result in the pipeline breaker. The result of the selective re-enumeration can be a different query execution plan, which in any case contains the already calculated intermediate results as intermediate result operators.

STEP 7 – PLAN SWITCH The final step in the loop replaces the current query execution plan in the adaptive query execution engine with the query execution plan from the selective re-enumeration, so that the next loop iteration can continue with identifying, instantiating, and executing the next pipeline.

The decision whether we continue with the current plan or switch to a new plan can have some implications on the build side of the hash joins, which are in most cases the pipeline breakers in our adaptive query execution engine. We further discuss these implications in the design of our extended CHT hash join in Section 5.4.2.

5.4.2 *Adaptive and NUMA-aware CHT building*

To keep the risk of query execution time degradations low, our adaptive query execution strategy re-optimizes only in pipeline breakers, such as the build side of a hash join. Plan switches between the build and the probe phase of a hash join introduce the following trade-off: Should the hash table be built, which could be wasted query execution time if the query execution plan changes, or should the build of the hash table be deferred. Deferring the build means that the pipeline breaker just collects the batches without building the actual hash table. The disadvantage is the loss of cache locality. To build the hash table, each batch has to be read again from main-memory, which can be a considerable overhead. For a CHT, which has to see the data twice, deferring the build procedure causes two entire reads from main-memory. So on the one hand it could be reasonable to always build the hash table right when the batches are pushed to the build side of the hash join. On the other hand, there are examples like the one we explain in Section 5.3: Multiple build sides in the initial plan in Figure 5.2 are significantly larger than the corresponding probe sides. A re-optimization as illustrated in Figure 5.3 switches the build and probe sides, and discards the hash table.

To make a decision for the illustrated trade-off, we consider another design dimension of parallel hash joins. We explain in Section 2.5.3.2 that building a hash table in parallel can be achieved by synchronization or partitioning. Synchronization means that there is only one hash table, and a synchronization mechanism, such as a mutex, which ensures that only one task at a time inserts into the hash table. For a large number of parallel tasks, the disadvantage is that a synchro-

nization mechanism such as a mutex introduces a bottleneck. Partitioning the hash range and building a hash table for each partition can avoid bottlenecks. But there can still be concurrent writes within a partition that have to be synchronized. Especially if there are heavy hitters within a partition, synchronization through mutexes can still create bottlenecks. To avoid synchronization within a partition, the build side of our CHT hash join is split. First, there is a dedicated partitioning phase, where each task can add tuples to any partition without any synchronization. Second, the actual build phase, where a CHT hash table is built for each partition by reading all tuples of the corresponding partition twice. We explain the standard CHT build procedure in Section 2.5.1.1. In sum, we spend another memory pass to avoid synchronization, and have to see the input three time: (1) to partition the data, (2) to fill the prefix bitmap of the CHT, and (3) to create the dense payload table of the CHT. At the same time, the dedicated partitioning phase provides us a nice re-optimization point.

In this section, we illustrate in detail how we improve the parallelism in the build phase of the CHT hash join, and integrate NUMA-awareness and a re-optimization point for adaptivity. We explain in Section 2.7.2 that a pipeline can start on a base table, which is horizontally partitioned into NUMA-allocated table partitions. As illustrated in Figure 2.5, each table partition consists of column partitions, which are also NUMA-allocated. Figure 2.6 illustrates the pipelining. At the beginning of a pipeline, there is a task for each table partition. The task creates a batch of tuples out of the table partition, by referencing the required column partitions in the batch. The task furthermore determines the tuple visibility. Next, the task pushes the batch through the pipeline to the pipeline breaker, which could be the build side of the CHT hash join. Like we illustrate it in Figure 2.6, the build side of the CHT hash join consists of two components: a generic pipeline breaker that just collects the batches, and the actual CHT that references the tuples that are stored in the generic pipeline breaker. Consequently, each batch that is pushed to the build side of our CHT hash join is first of all added to the generic pipeline breaker, before it is forwarded to the CHT build process. The entire build process is illustrated in the Figures 5.6 and 5.7, and has three phases: (1) partitioning, (2) adaption, (3) building. Figure 5.6 illustrates the end of the pipeline i with the generic pipeline breaker, and our CHT partitioning phase. Furthermore, Figure 5.6 shows the tasks T_0 to T_7, which push the batches B_0 to B_7 through pipeline i, and add the batches to the generic pipeline breaker. The batches contain the column partitions of column C, on which the CHT is built. Figure 5.6 also shows the two NUMA-nodes N_0 and N_1 where the tasks run, and where the batches, column partitions, and CHT partitions are allocated. There can be of course more batches, tasks, and NUMA-nodes than illustrated.

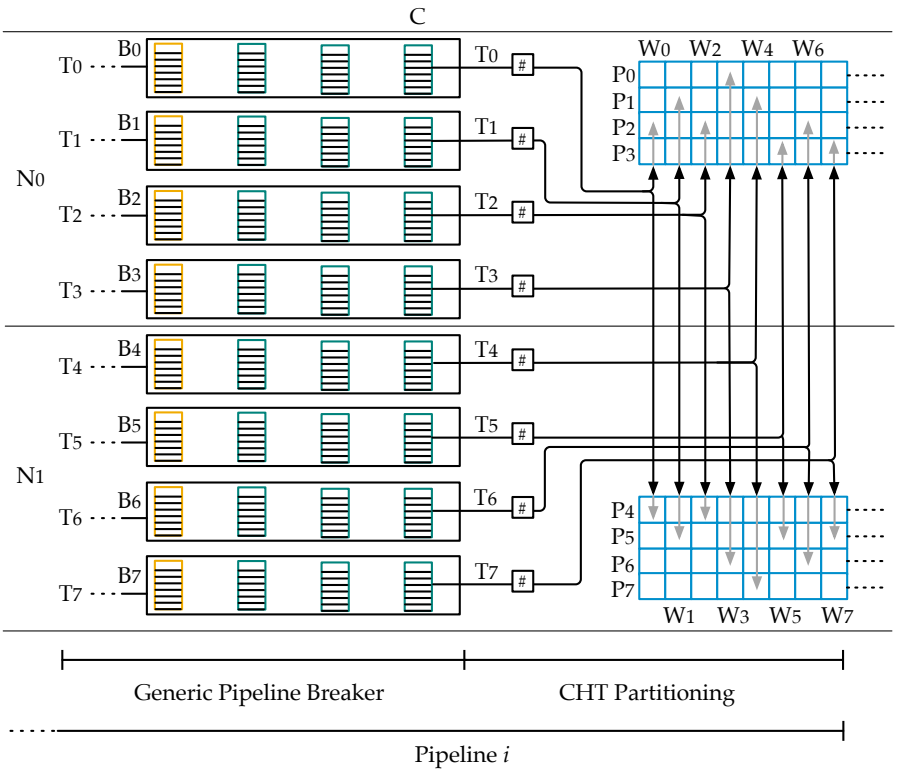


Figure 5.6: Illustration of NUMA-aware CHT partition procedure.

PARTITIONING PHASE In the partitioning phase of our CHT build, each task of pipeline i iterates over the entries in their column partition of Column C. Each entry of a column partition is a key in the CHT. The payload entries in the CHT are the identifiers of the tuples. A tuple identifier consists of a reference to the batch in the generic pipeline breaker, illustrated in Figure 5.6, and the row of the tuple in the batch. As a first step, each task calculates the hash value of each key, i.e., entry in its column partition, to determine its CHT partition. In Figure 5.6 there are eight CHT partitions P_0 to P_7. Next, the tasks write the key/payload-pairs to a container that collects all the entries of one CHT partition. Since multiple tasks can write key/payload-pairs to one CHT partition container concurrently, the access has to be synchronized. To avoid synchronization, each worker thread of our NUMA-aware task scheduler has its own partition in each CHT partition. Figure 5.6 shows partitions for eight worker threads W_0 to W_7. For load balancing reasons in the third phase, the number of CHT partitions P should be a multiple of the number of worker threads W in the NUMA-aware task scheduler. The partitioning phase is finished when each task has processed every key in their column partition.

ADAPTION PHASE This is the perfect point for a re-optimization and potentially a query execution plan switch. All parallel tasks that execute the pipeline i are finished. In addition, the generic pipeline

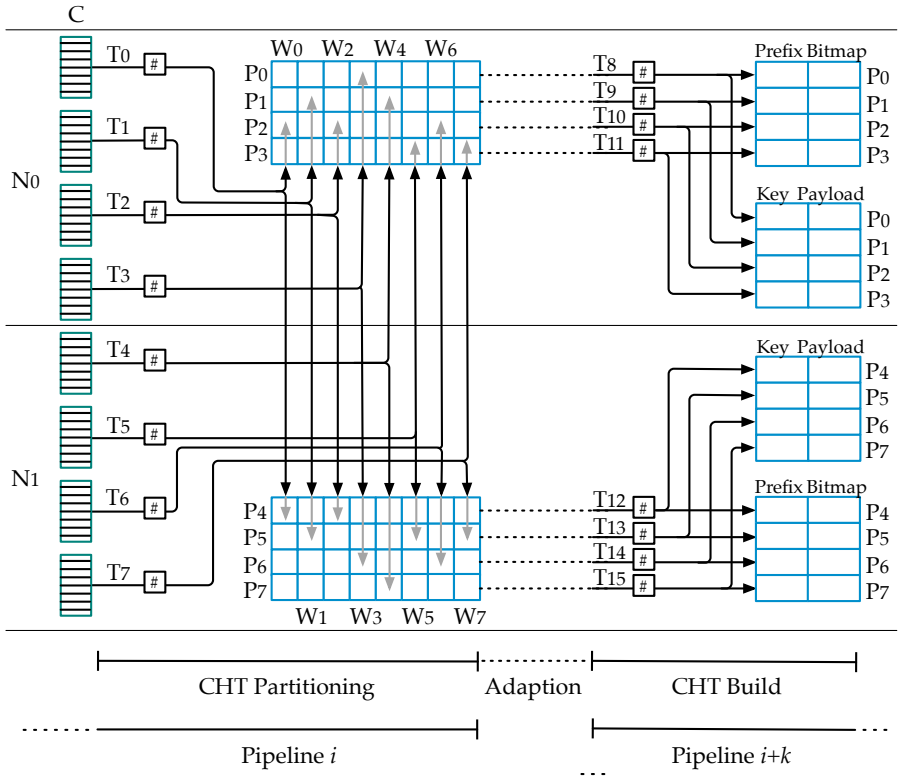


Figure 5.7: Illustration of adaptive and NUMA-aware CHT build procedure.

breaker on the left-hand side of Figure 5.6 contains a complete intermediate result, and its true cardinality. Furthermore, there is a hash-partitioned set of key/payload-pairs, illustrated on the right-hand side of the Figure 5.6. As illustrated in Figure 5.7, our adaptive query processor can perform a re-optimization. After a re-optimization there are four cases:

1. There is no plan switch, so that the CHT partitioning phase in pipeline i is followed by the CHT build phase in pipeline $i+k, k \geqslant 1$. The CHT build phase is a preparation step of pipeline $i+k, k \geqslant 1$, which contains the corresponding CHT probe side.

2. There is a new plan, which re-orders the operators, so that the CHT partitioning phase is performed as preparation step in a much later executed pipeline $i+k, k > 1$.

3. There is a new plan, which re-orders the operators, so that the intermediate result in the generic pipeline breaker of pipeline i is joined on a different column.

4. There is a new plan, and the intermediate result in the generic pipeline breaker of pipeline i becomes a probe side.

In the two latter cases, the hash-partitioned set of key/payload-pairs illustrated on the right-hand side of Figure 5.6 is discarded. Consequently, the only additional overhead is the partitioning phase, which

is considerably cheaper than a complete CHT build. When the plan changes, and the join is performed on a different column, the CHT partitioning has to be repeated. When a build side becomes a probe side, a new pipeline is created together with new tasks, which push the batches from the generic pipeline breaker, illustrated in Figure 5.6, through the new pipeline. Although the partitioning effort is wasted in the two latter cases, it can most probably be compensated through the better query execution plans. In the first two cases, our adaptive query execution engine continues at some point with the third phase of our CHT build.

BUILD PHASE The third phase of our CHT build is executed as a preparation step of pipeline $i + k$, which contains the corresponding CHT probe side. As we explain in Section 5.4.1, query execution plans are executed pipeline by pipeline, and the next pipeline to be executed is determined just in time in the current plan. Before the first batch is pushed through a pipeline, the query execution engine checks if there are CHT hash join probe sides, and triggers the third phase of the CHT build for the corresponding build sides.

The third phase of our CHT build is illustrated on the right-hand side of Figure 5.7. It is the actual CHT build phase, in which a local CHT is built for each CHT partition. The building of a local CHT is identical to building a standard CHT, which we explain in Section 2.5.1.1. To build the local CHTs, our adaptive query execution engine creates one task per CHT partition. In Figure 5.7, those are the tasks T_8 to T_{15}. In order to balance the load of the parallel tasks in this phase, there should be more CHT partitions than worker threads in the NUMA-aware task scheduler. To cover the case of uniform-sized CHT partitions, the number of CHT partitions should be a multiple of the number of worker threads in the NUMA-aware task scheduler.

To sum it up, the build side of our CHT hash join consists of a generic pipeline breaker, which collects all batches that are pushed to it. On top, we build our partitioned CHT, which references the tuples in the generic pipeline breaker. The generic pipeline breaker design has an additional advantage: There are situations caused by plan switches, where phase three of the CHT build is not triggered anymore, because the intermediate result becomes a probe side in the adapted plan, or the next join of the intermediate result is on a different column. Since the generic pipeline breaker collects all batches that are pushed to it, and does not modify them, we can also build a new CHT on a different column. The generic pipeline breaker also enables to continue the original pipeline by creating a task per batch, which pushes the batch through the continued pipeline.

Such a situation is illustrated in the motivating example in Section 5.3. The initial plan in Figure 5.2 is re-optimized after joining the

tables R,S, and T. The re-optimization takes place in the build side of the join between RST and U. Consequently, the partitions for the CHT on RST are already created. In the adapted plan in Figure 5.3, RST becomes the probe side in the join with VUW, so that there is no hash table build on RST because the pipeline was continued.

5.4.3 *Selective Re-Enumeration*

Next, we illustrate the selective re-enumeration algorithm, to speed-up re-optimizations. The selective re-enumeration is Step 6 of our adaptive query execution strategy in Figure 5.4. We start with a brief recap of the running example of Figure 5.5. The Steps 1 to 3 of our adaptive query execution strategy selected, instantiated, and executed the pipeline starting on base table S. The pipeline ends in the build side of RS $\bowtie$ T. The generic pipeline breaker on the build side contains the intermediate result RS. In Step 4, we assume the re-optimization criterion, which we further explain in Section 5.4.4, decided to re-optimize. According to Figure 5.4, Step 5 follows. In Step 5, the CHT and the generic pipeline breaker, which contains the intermediate result RS, are wrapped into an intermediate result operator, and referenced in the plan table as optimal plan of plan class RS. At the same time, the estimated cardinality of plan class RS is replaced by its true cardinality, which was derived from the generic pipeline breaker. After the intermediate result is referenced in the plan table, and the cardinality updated, Step 6, the selective re-enumeration can start.

5.4.3.1 *Example*

The selective re-enumeration is based on the DPsize enumeration algorithm, which we explain in Section 2.6.1. Before we formalize the selective re-enumeration in Algorithm 5.1, we discuss the running example. Figure 5.8 shows the plan classes and plans, which the dynamic programming optimizer enumerates to identify the estimated optimal query execution plan in Figure 5.5. For simplicity, Figure 5.8 contains only logical query execution plans. Furthermore, we assume the optimizer keeps only one optimal plan per plan class and no additional plans with different physical properties. This is no limitation, because as conventional dynamic programming optimization, the selective re-enumeration can of course support multiple plans per plan class with different physical properties. In dynamic programming optimization, an optimal solution can always be constructed from optimal sub-solutions. Figure 5.8 shows the optimal plan of each plan class in bold font. The optimal plans are combined to create new plans, as illustrated by dotted lines. Figure 5.8 also shows the optimal query execution plan of Figure 5.5 in solid bold lines.

In order to guarantee a correct re-optimization result, all plans that directly or indirectly reference the plan class, which was updated

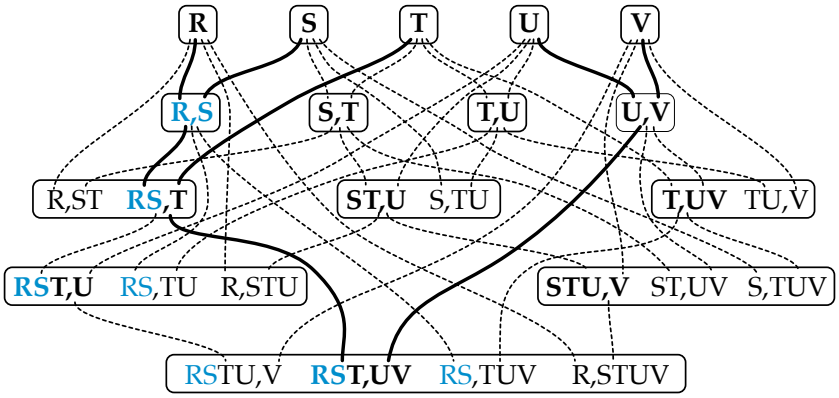


Figure 5.8: Overview of the plan classes and logical query execution plans that our dynamic programming optimizer enumerates to identify the estimated optimal query execution plan in Figure 5.5.

by Step 5 of our adaptive query execution strategy in Figure 5.4, have to be re-enumerated. Selective re-enumeration has similarities to the optimality ranges calculation we present in Section 3.2. The optimality range calculation also considers all plans, which directly or indirectly reference an intermediate result with varying cardinality. Figure 5.9 categorizes the plan classes of the example in Figure 5.8. There are first of all plan classes in which some of the plans have to be re-enumerated, because they directly or indirectly reference the plan class RS. In Figure 5.8, those are the plans in which RS is highlighted. Second, Figure 5.9 shows plan classes, which became obsolete through the execution of RS. Plans and plan classes can become obsolete, because we only re-optimize the not-yet-executed query parts to avoid wasting already calculated intermediate results. Executing a plan of an obsolete plan class in any case results in wasting calculated intermediate results. Therefore, we do not consider these plan classes in the selective re-enumeration. Since we assume that the plan table is not needed after the query execution, we remove all obsolete plan classes from the plan table before the selective re-enumeration. In case the plan table has to be kept for future query executions, it is of course possible to keep the plan table untouched. In this case, the check whether a plan class is obsolete has to be performed during the re-enumeration. Third, there are plan classes, which are completely unaffected by the cardinality update of RS. Re-enumerating those plan class is redundant work. Consequently, our selective re-enumeration does not re-enumerate the plans in the unaffected plan classes. In contrast to our selective re-enumeration algorithm, a full re-optimization enumerates plans of unaffected plan classes.

5.4.3.2 *Formal Algorithm Description*

Apart from the implementation of the enumeration algorithm, the selective re-enumeration shares all implementations with a conven-

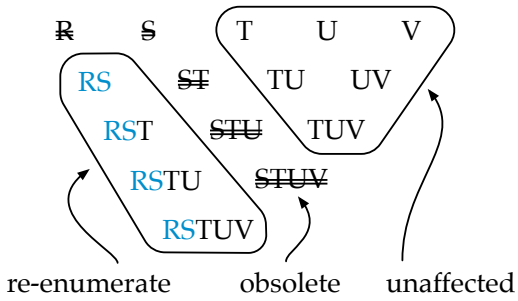


Figure 5.9: Categorization of the plan classes of the example in Figure 5.8.

tional dynamic programming query optimizer. We formalize the selective re-enumeration in Algorithm 5.1. The first input in Line 1 is a reference to the original plan table, which is a container of plan classes. Within the container, the plan classes are ordered by their size, i.e., the number of base tables they reference. Our adaptive query execution engine keeps the original plan table after the initial plan search. The second input are the join predicates, and the third input is the identifier of the executed plan class. Our selective re-enumeration algorithm is a modified DPsize enumeration algorithm, which works on the original plan table. Other dynamic programming enumeration algorithms, such as DPsub [27] or DPccp [44] can be also modified for selective re-optimizations. Like in the example in Figure 5.9, the selective re-enumeration initially removes in Line 2 all plan classes from the original plan table, which became obsolete through the executed plan class. The first difference to a conventional DPsize is a temporary container `tempPCs` in Line 5, which collects the identifiers of re-enumerated plan classes, categorized by size. The initial size that is added to `tempPCs` is the size of the executed plan class in Line 6. For the size of the executed plan class, Line 7 adds the identifier of the executed plan class. Next, the entries in `tempPCs` are combined with plan classes from the original plan table, to create new plans that are in turn added to `tempPCs`.

The loop in Line 9 iterates over all plan class sizes, from the size of the executed plan class to the size of the final plan class. The next nested loop in Line 12 together with the instruction in Line 14 enumerates all plan class size combinations, which create plans of the current size. Each combination consists of outer and inner plan class size (Lines 12 and 14). The nested loops in the Lines 16 and 17 iterate over all outer and inner plan classes with a corresponding size to enumerate all possible combinations of corresponding outer and inner plan classes. Compared to the conventional DPsize algorithm, the outer plan class in Line 16 of the selective re-enumeration is always a plan class from `tempPCs`. Consequently, the outer plan class directly or indirectly references the executed plan class or is the executed plan class. For the same reason, the loops in the Lines 9 and 12 start at the executed plan class size plus 1 or executed plan class size. In the con-

Algorithm 5.1 Selective Re-Enumeration

```
1: function selectiveReEnumerate(Reference⟨PlanClassContainer⟩
   planClasses, List⟨JoinInfo⟩ joins, BitSet executedPlanClassId)
2:     removeOblsoletePCs(planClasses, executedPlanClassId)
3:     declare Int exSize ← executedPlanClassId.countBits()
4:     declare Int maxSize ← planClasses.finalPcId().countBits()
5:     declare PlanClassIdsBySizeContainer tempPCs
6:     tempPCs.addFirstSize(exSize)
7:     tempPCs.withSize(exSize).add(executedPlanClassId)
8:     // iterate over plan class sizes
9:     for pcSize = exSize + 1; pcSize < maxSize; pcSize++ do
10:        tempPCs.addNextSize()
11:        // enumerate outer plan class sizes
12:        for outSize = exSize; outSize < maxSize; outSize++ do
13:            // derive inner plan class size
14:            declare Int inSize ← pcSize − outSize
15:            // enumerate plans by combining plan classes
16:            for each outPC in tempPCs.withSize(outSize) do
17:                for each inPC in planClasses.withSize(inSize) do
18:                    // search for joins that connect plan classes
19:                    declare List⟨JoinInfo⟩ foundJoins
20:                    foundJoins ← findJoins(outPC, inPC, joins)
21:                    if !foundJoins.empty() then
22:                        // fetch plan class of identified plan
23:                        declare BitSet pcId = outPC.id or inPC.id
24:                        declare Reference⟨PlanClass⟩ planClass
25:                        planClass ← planClasses.find(pcId)
26:                        if tempPCs.lastSize().tryInsert(pcId) then
27:                            // first time re-enum touches plan class
28:                            planClass.clearOptimalPlan()
29:                        end if
30:                        planClass.add(outPC, inPC, foundJoins)
31:                    end if
32:                end for
33:            end for
34:        end for
35:    end for
36: end function
```

ventional DPsize algorithm, these loops start at size 1. The inner plan class in Line 17 is always a plan class from the original plan table.

Since we do not allow cross products, Line 20 searches for each enumerated combination of outer and inner plan class if there is at least one join predicate, which connects the two plan classes. If one or more join predicates are found, Line 23 continues with deriving the plan class identifier of the enumerated plan. Next, Line 26 fetches

a reference to the plan class in the original plan table. In case the current re-optimization touches the plan class of the enumerated plan for the first time, Line 28 removes the old optimal plan from the plan class. This is crucial to avoid the following issues: The re-optimized plan of a plan class, which is based on up-to-date cardinalities, cannot be compared to an old optimal plan of the same plan class. Plans can become more expensive in the re-enumeration because the true cardinality of an intermediate result was underestimated. If the old optimal plan is not removed from the plan class, the old optimal plan can be referenced in the re-optimized plan, so that the re-optimized plan is not necessarily optimal with respect to the given inputs. In addition, when the re-optimized plan references the old plan, already calculated intermediate results are ignored and calculated again.

The check if a re-optimization touches a plan class for the first time is realized through the insertion into tempPCs in Line 26. In any case, the enumerated plan is specified by outer and inner plan class as well as some join attributes that connect the plan classes. The enumerated plan is proposed through a conventional DPsize procedure in Line 30 to the plan class. After the invoked procedure in Line 30 estimated the cost of the enumerated plan, it might set it as optimal plan in the plan class, if it is the first plan or cheaper than the existing plan.

After the selective re-enumeration finishes, the new optimal plan is the cheapest plan of the final plan class in the original plan table. From the logical point of view, the re-optimized plan could be the same plan as the old plan or a different plan. Even if it logically is the same plan, the re-optimized plan references the executed plan class as intermediate result operator, containing the generic pipeline breaker and a pre-partitioning for a CHT. Step 7 of our adaptive execution strategy in Figure 5.4 sets the re-optimized plan as current optimal plan in the query execution engine. In one of the next pipeline instantiations, the intermediate result operator will be detected, so that the intermediate result will be re-used, and not get lost.

5.4.4 *Re-optimization Criteria*

Left open in our adaptive query execution strategy in Figure 5.4 is Step 4, the re-optimization criteria. As soon as a pipeline is executed, and the pipeline breaker contains the intermediate result, a selective re-optimization could be triggered to search for a better plan.

The cheapest, but also the least precise criterion are heuristically defined optimality ranges based on the estimated cardinality [31, 41]. An uncertainty value U is assigned to an estimated cardinality $\hat{f}$, so that the lower bound of the optimality range $f^{\downarrow} = E - (E * 0.1 * U)$ and the upper bound $f^{\uparrow} = E + (E * 0.2 * U)$. Our experiments on precise optimality ranges in Section 3.3.1 show that heuristics fail to characterize optimality ranges. Optimality ranges are independent of the

estimate. The estimates can be close to the lower or upper bound of an optimality range, and also the width of optimality ranges is independent of the estimate. Our experiments in Section 3.3.1 also show that already small cardinality deviations can result in a cheaper plan.

We argue that missing a better query execution plan is considerably worse compared to some redundant re-optimizations, in which no better plan is found. Corresponding to our problem statement in Section 5.2, we only avoid a re-optimization, when we can guarantee that there is no better plan. According to our experiments on optimality ranges in Section 3.3.1, already an estimation error of a few tuples can result in a different optimal plan. We conclude that we can only guarantee that there is no better plan, if there is no estimation error. Consequently, we always trigger a selective re-enumeration when an intermediate result in a pipeline breaker has an estimation error of one or more tuples. The only limitation is that there have to be two or more operators left in the query, because there are no operator order alternatives if only one operator is left. Since we only re-optimize in pipeline breakers, the number of re-optimizations is at most the number of pipeline-breaking operators in the plan minus 2. Furthermore, we demonstrate below that a selective re-enumeration enumerates a considerably smaller number of plans compared to a full re-optimization. Our experiments in Section 5.5 further reveal that the additional re-optimization effort is low, and can be compensated through faster plans. Another option for the re-optimization criterion are precise optimality ranges, calculated by our algorithm in Section 3.2. We experimentally show next that optimality range calculation has to enumerate at least the number of plans, which a selective re-enumeration has to enumerate to make the same decision. Since the selective re-enumeration also identifies the better plan, other than an optimality range, the selective re-enumeration is the superior re-optimization criterion for ad-hoc queries compared to optimality ranges. The complexity of optimality range calculation can only amortize in multiple executions of the same query.

To illustrate the computational complexity of the three re-optimization criteria full re-optimization, selective re-enumeration, and optimality ranges we do the following experiment. We compare the number of enumerated plans, i.e., feasible plans or connected sub-graph complement pairs, for the three re-optimization criteria. The experimental setup is the same as in the experiment on the number of enumerated plans for optimality range calculation in Section 3.3.2, except that we consider only one chain query, which joins 20 tables. As in the experiment in Section 3.3.2, we randomly create 100 databases on which the chain query runs. For each of the 100 random databases, we search the optimal query execution plan for the chain query. In the experiment, we assume we need a re-optimization decision on the base table edges in the query execution plans. Base table edges

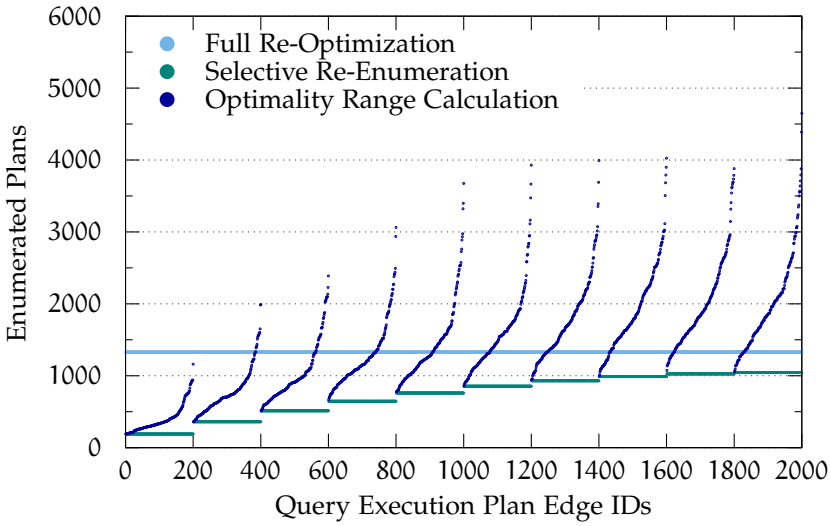


Figure 5.10: Comparing re-optimization criteria on base table edges of 100 random chain queries with 20 Tables.

are the worst case in terms of enumeration effort. Since each of the 100 query execution plans joins 20 tables, there are 2000 base table edges. For each of the edges, we compare the number of enumerated plans of a full re-optimization, a selective re-enumeration, and an optimality range calculation. Figure 5.10 shows the results.

The 2000 query execution plan edges are plotted along the x-axis, and the number of enumerated plans along the y-axis. The edges on the x-axis are ordered by the number of enumerated plans of the full optimization, followed by the selective re-enumeration and the optimality range calcualtion. The number of enumerated plans of a full re-optimization is constant over all base table edges. In our experiments each full optimization in our dynamic programming optimizer enumerates 1330 logical query execution plans including bushy trees. This the same number, which Ono and Lohman [14] report for exactly that case, i.e., feasible, logical, and bushy query execution plans without cross products enumerated by dynamic programming for chain queries joining 20 tables. The number of enumerated plans for the described case can also be calculated through Equation 3.21.

The number of enumerated plans for the calculation of one optimality range varies. As we explain in Section 3.3.2, the variation comes from the structure of the query execution plan, e.g., left-deep, bushy, or something in between, and the effectiveness of our pruning strategy. There are some edges where the optimality range calculation enumerates less plans compared to a full re-optimization, but there are also edges where a full re-optimization performs better.

The results of the selective re-enumeration also vary for different query execution plan structures. The selective re-enumeration has to enumerate more plans when a base table edge is deeper in the plan,

compared to base table edges that are higher in the plan. In the experiment, the selective re-enumeration always enumerates a smaller number of plans compared to the full re-optimization. The experiment also shows that the number of enumerated plans by selective re-enumeration are a lower bound for the number of enumerated plans by optimality range calculation. The selective re-enumeration enumerates for each considered edge the smallest number of plans, and is therefore the superior re-optimization criterion. Compared to the optimality range calculation, the selective re-enumerations furthermore returns the new plan. In case an optimality range decides to re-optimize, there is the additional effort to search the new plan.

5.5 EXPERIMENTAL EVALUATION

To evaluate our adaptive query processor, we choose a similar experimental setup as in the evaluation of the robustness metrics and robust plan selection in Section 4.6. Identical are the considered benchmarks, the baseline, the system, the kernel, and the compiler. Consequently, all execution times are comparable to the results in Section 4.6.

The baseline in our experiments is again the estimated optimal query execution plan, which is selected in the initial query optimization solely based on estimations. We argue in Section 4.6 that our join optimizer's choice of the estimated optimal plan is very similar to the choice of popular commercial and free database systems, for the considered join queries. The reason is that our join optimizer relies on dynamic programming [4], such as DB2 [17] and Postgres [78]. As Postgres, it exhaustively searches the plan space including bushy trees. Section 2.7 shows that its cardinality estimator is competitive. Furthermore, we use the C_{mm} [93] cost function, which has a strong correlation to our main-memory execution engine, according to Section 2.7. We denote the execution of the estimated optimal plan as *conventional execution*. Accordingly, the query execution in our adaptive query processor is denoted as *adaptive execution*, and the corresponding query execution plan as *adaptive plan*.

We furthermore compare the adaptive execution against another execution strategy, whose re-optimization criterion is based on heuristically defined optimality ranges. According to Kabra and DeWitt [31] and Babu et al. [41] an optimality range can be defined as follows: Based on an uncertainty value U and an estimate E, the lower bound of an optimality range for cardinality $f^{\downarrow} = E - (E * 0.1 * U)$ and the upper bound $f^{\uparrow} = E + (E * 0.2 * U)$. Section 3.3.1 illustrates that this heuristic fails to characterize optimality ranges. In the following experiments the uncertainty value U is 500. We denote the query execution using the heuristic re-optimization criterion as *heuristic execution* and the corresponding query execution plan as *heuristic plan*. The heuristic execution strategy is similar to our adaptive execution

strategy in Section 5.4.1, but more conservative. It only triggers a re-optimization when the true cardinality is outside of the heuristically defined optimality range. In contrast, our adaptive execution strategy in Section 5.4.1 triggers re-optimizations on cardinality estimation errors of one tuple. Since we implemented the heuristic execution strategy in our adaptive query processor, it uses the extended CHT hash join implementation, and the selective re-enumeration.

Next to the estimated optimal plan and the heuristic plan, we compare each adaptive plan against the fastest plan. As in Section 4.6, the fastest plan is identified in an offline analysis through the execution of the 500 estimated cheapest plans. Note that the fastest plan is no competitive approach to the estimated optimal plan, the heuristic plan, or the adaptive plan, because its calculation can take several hours. Consequently, it is not applicable at query optimization time. Searching the fastest plan reduces the impact of cardinality estimation errors, and cost function inaccuracies. Nevertheless, there can be an even faster plan, which is not in the 500 estimated cheapest plans.

Corresponding to the problem statement, Section 5.5.1 evaluates the true cost improvements of the heuristic plans, adaptive plans, and fastest plans with respect to the estimated optimal plans. Section 5.5.2 illustrates the resulting end-to-end query execution times, and speedups with respect to conventional plan execution. We compare the number of re-optimizations and plan switches in Section 5.5.3, and the resulting adaption effort in Section 5.5.4. The numbers we report in Section 5.5.1 and 5.5.3 do not depend on the machine the experiments run on. Reported execution times are comparable to the query execution times in Section 4.6, because they were taken on the same two socket Intel Xeon E5-2660 v3 system with 128 GB of main memory, running the same Linux 4.4.120 kernel. As we explain in Section 2.7, our main-memory query execution engine performs join operators as hash joins. The query optimizer including the selective re-optimization are single-threaded. Corresponding to the experiments in Section 4.6, the entire query processor is compiled with `gcc` version 7.2.0 and optimization option -O3.

To enable a comparison with the experimental results on the robustness metrics and robust plan selection, we choose exactly the same benchmarks as in the experiments in Section 4.6. The first benchmark is based on the Join Order Benchmark (JOB) [78]. The JOB uses the real-world database from IMDb with skew, correlations, and different join relationships that cause estimation errors. We modified the original queries to be pure join queries, which results in 33 complex queries containing cycles and multiple join conditions between subplans. Since pure join queries without any filters on base tables create large intermediate results, we use the `movie_id` column as scale factor for the benchmark. We limit the `movie_id` column of all tables to values less equal than 100,000. In the end, the scale factor enables to run

31 different queries. We argue that the scale factor does not limit the validity of our results, because it creates a snapshot of the database at the point when it contained only 100,000 movies.

As in Section 4.6, the second benchmark is synthetic with generated data and join queries. We consider again the query topologies: chain, cycle, star, and snowflake. All topologies join 10 tables. The snowflake topology has a fact table with three dimension tables. Each dimension table has again two sub-dimensions. For each topology, we create one query and 100 different data sets. Furthermore, we generate 100 queries with a random topology and a corresponding data set. The random topology creator starts with a random connected query graph, and adds with a probability of 4% additional edges to create cycles. The random topologies also join 10 tables. For all generated data sets, the base table cardinalities are uniform random numbers between 10,000 and 100,000. The data sets contain skew and arbitrary correlations between columns to generate expanding and selective joins. There are foreign key and m:n join relationships. The join cardinalities between two base tables R_i and R_j are uniform random numbers between $max(|R_i|, |R_j|) - 5000$ and $max(|R_i|, |R_j|) + 5000$.

Equivalent to Section 4.6, we denote the conventional execution of the estimated optimal plan throughout the experiments as EO, and the fastest plan as FA. Furthermore, we denote the heuristic execution and plan as HE, and the adaptive execution and plan as AE.

5.5.1 *Improvement of True Cost*

Since true cardinalities are superior to estimated cardinalities, considering true cardinalities in the plan selection should result in better or at least equally good query execution plans. According to the formal problem description in Section 5.2, a precondition for query execution time improvements is that the adaptive plan AE has smaller true cost $\mathring{c}$ compared to the initial or estimated optimal plan EO. To measure the true cost improvement of an arbitrary plan P with respect to the estimated optimal plan EO, we calculate the true cost improvement factor $\mathring{c}_{imp}$ between the true cost of the estimated optimal plan $\mathring{c}_{EO}$, and the true cost of plan P, $\mathring{c}_P$:

$$\mathring{c}_{imp} = \begin{cases} \mathring{c}_{EO}/\mathring{c}_P & \text{if } \mathring{c}_{EO} \geqslant \mathring{c}_P \\ -\mathring{c}_P/\mathring{c}_{EO} & \text{otherwise} \end{cases} \quad \text{where} \quad \mathring{c}_P > 0 \quad \text{and} \quad \mathring{c}_P > 0$$

Consequently, a positive $\mathring{c}_{imp}$ reveals the true cost improvement of plan P with respect to the estimated optimal plan EO, and a negative $\mathring{c}_{imp}$ a true cost degradation. Figure 5.11 shows some typical results for the Join Order Benchmark. The typical results are the nine worst queries and the nine best queries with respect to the speedup of end-to-end query execution time of the adaptive plan AE relative to the

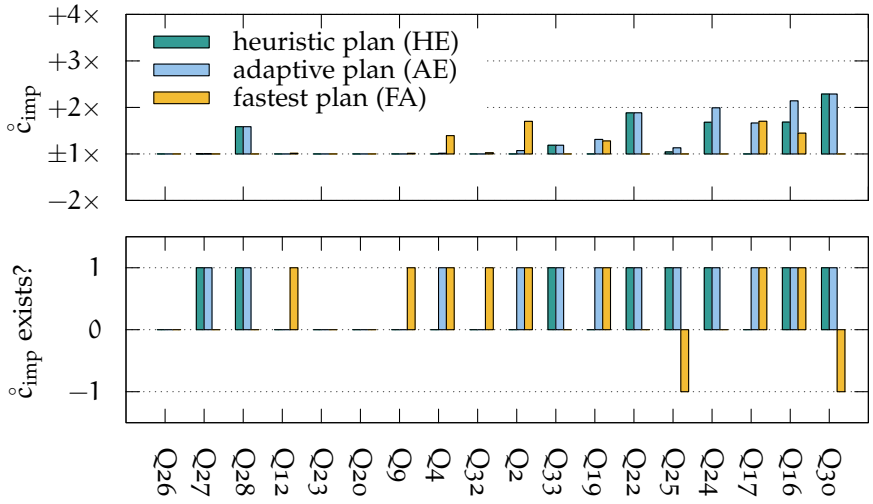


Figure 5.11: Typical results of experiment comparing the true cost improvement factor relative to the estimated optimal plan (EO) on Join Order Benchmark queries.

estimated optimal plan EO. Throughout the experiments, we always consider the same typical queries. The queries are plotted along the x-axis, and ordered by increasing speedup of the end-to-end query execution time. The y-axis shows the true cost improvement $\mathring{c}_{imp}$, and whether there is a positive, negative or zero $\mathring{c}_{imp}$ value. For the Join Order Benchmark queries in Figure 5.11, the heuristic plans HE and the adaptive plans AE achieve the best $\mathring{c}_{imp}$ with $+2.29\times$ in Q30. The best $\mathring{c}_{imp}$ for the fastest plans FA is achieved in Q2 and Q17 with $+1.70\times$. In the Join Order Benchmark, there is no case for the heuristic plans HE and adaptive plans AE with a negative $\mathring{c}_{imp}$. For the fastest plans FA, only Q25 and Q30 have a slightly negative $\mathring{c}_{imp}$. Furthermore, the heuristic plans HE improve the true costs, i.e., have a positive $\mathring{c}_{imp}$ in 11 out of 31 queries, the adaptive plans AE in 16 out of 31 queries, and the fastest plans FA in 19 out of 31 queries. The adaptive plans dominate the average $\mathring{c}_{imp}$ with $+1.47\times$, compared to the heuristic plans with $+1.33\times$, and the fastest plans with $+1.06\times$.

Figure 5.12 shows some typical results of the synthetic benchmark queries with a random query topology. From the 100 queries in the benchmark, we again picked the nine worst and the nine best queries with respect to the speedup of end-to-end query execution time of the adaptive plan AE relative to the estimated optimal plan EO. The adaptive plans AE dominate the best $\mathring{c}_{imp}$ with $+3.90\times$ in Q62, compared to the heuristic plans HE with $+2.86\times$ in Q10, and the fastest plans FA with $+2.73\times$ in Q62. There is one query, Q64, where the adaptive plan has a negative $\mathring{c}_{imp}$ of $-1.01\times$. In this case, better knowledge, i.e., about true cardinalities, causes a worse plan quality. The root cause is an estimation error during the re-optimization. Cases where better knowledge results in a worse plan quality are infrequent, but can oc-

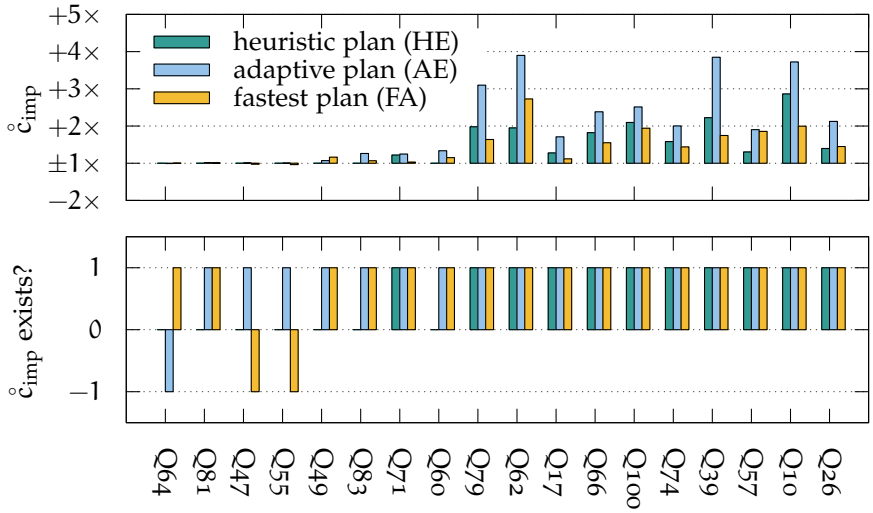


Figure 5.12: Typical results of experiment comparing the true cost improvement factor relative to the estimated optimal plan (EO) on synthetic benchmark queries with random topology.

cur. We explain in Section 7.1 how they can be further reduced. Nevertheless, the adaptive plans AE achieve a positive $\mathring{c}_{imp}$ in 99 out of 100 queries, and dominate the average $\mathring{c}_{imp}$ with $+1.56\times$. The heuristic plans HE in contrast improve the plan quality in 40 out of 100 queries, and achieve an average $\mathring{c}_{imp}$ of $+1.26\times$. The fastest plans FA perform worse compared to the heuristic and adaptive plans AE with respect to $\mathring{c}_{imp}$. They face a true cost degradation in 4 queries, and have the strongest true cost degradation with $-1.10\times$. Although the fastest plans AE improve the true costs in 96 out of 100 queries, they achieve the smallest average $\mathring{c}_{imp}$ with $+1.25\times$.

Table 5.1 summarizes the results of all benchmarks through best, worst, and average $\mathring{c}_{imp}$, as well as the number of queries that achieved positive and negative $\mathring{c}_{imp}$ values. Compared to the heuristic plans HE and the fastest plans FA, the adaptive plans AE achieve in each benchmark the largest true cost improvement $\mathring{c}_{imp}$. The best $\mathring{c}_{imp}$ over all benchmarks is achieved by AE in the random query benchmark with $+3.90\times$. Furthermore, the adaptive plans AE improve in each synthetic benchmark the largest number of queries, i.e., queries with a positive $\mathring{c}_{imp}$ value. In each synthetic benchmark, the adaptive plans AE improve at least 95 out of the 100 queries. Queries where adaptive plans AE create a worse plan quality, i.e., a negative $\mathring{c}_{imp}$, are infrequent. The peak number of queries with a negative $\mathring{c}_{imp}$ is 4 out of 100 in the cycle benchmark. Compared to adaptive plans AE, the heuristic plans HE are identified through a more conservative re-optimization strategy. In our experiments, this conservative strategy always results in a positive true cost improvement $\mathring{c}_{imp}$ or no true cost improvement $\mathring{c}_{imp} = 1.00\times$. Furthermore, the heuristic plans HE improve a smaller

		best $\mathring{c}_{imp}$	worst $\mathring{c}_{imp}$	average $\mathring{c}_{imp}$	$+\mathring{c}_{imp}$	$-\mathring{c}_{imp}$
JOB	HE	$+2.29\times$	$1.00\times$	$+1.33\times$	11/31	0/31
	AE	$+2.29\times$	$1.00\times$	$+1.47\times$	16/31	0/31
	FA	$+1.70\times$	$1.00\times$	$+1.06\times$	19/31	2/31
Chain	HE	$+1.42\times$	$1.00\times$	$+1.03\times$	1/100	0/100
	AE	$+2.46\times$	$-1.05\times$	$+1.21\times$	95/100	2/100
	FA	$+2.30\times$	$-1.11\times$	$+1.17\times$	75/100	25/100
Cycle	HE	$+1.60\times$	$1.00\times$	$+1.01\times$	4/100	0/100
	AE	$+2.47\times$	$-1.12\times$	$+1.26\times$	95/100	4/100
	FA	$+2.46\times$	$-1.03\times$	$+1.29\times$	95/100	5/100
Star	HE	$+1.56\times$	$1.00\times$	$+1.02\times$	8/100	0/100
	AE	$+1.77\times$	$1.00\times$	$+1.04\times$	97/100	1/100
	FA	$+1.62\times$	$-1.03\times$	$+1.03\times$	88/100	11/100
Snowflake	HE	$+2.77\times$	$1.00\times$	$+1.07\times$	7/100	0/100
	AE	$+3.76\times$	$-1.02\times$	$+1.19\times$	99/100	1/100
	FA	$+2.56\times$	$-1.21\times$	$+1.09\times$	89/100	10/100
Random	HE	$+2.86\times$	$1.00\times$	$+1.26\times$	40/100	0/100
	AE	$+3.90\times$	$-1.01\times$	$+1.56\times$	99/100	1/100
	FA	$+2.73\times$	$-1.10\times$	$+1.25\times$	96/100	4/100

Table 5.1: Overview of experiments on the true cost improvement factor relative to the estimated optimal plan (EO) on different benchmarks, comparing the adaptive plans (AE) against the heuristic plans (HE), and fastest plans (FA).

number of queries, and in each benchmark they have a significantly smaller average $\mathring{c}_{imp}$ compared to the adaptive plans AE.

To sum it up, our experimental results reveal that information from the query execution, i.e., the true statistics can improve the average plan quality, because the adaptive plans AE and heuristic plans HE have mostly a positive true cost improvement $\mathring{c}_{imp}$ or no true cost improvement $\mathring{c}_{imp} = 1.00\times$. Comparing our adaptive plans AE with the heuristic plans HE shows that the adaptive plans improve the plan quality stronger and in more queries. The reason is that our adaptive execution strategy triggers more re-optimizations, so that more runtime feedback is included in the plan choice. Furthermore, the performance of the heuristic plans heavily depends on the uncertainty value for the heuristically defined optimality ranges. Throughout the experiment, we choose an uncertainly value of 500, which results in a quite competitive performance to the adaptive plans in the Join Order Benchmark. For the synthetic benchmark, the uncertainty value could

be smaller to trigger more re-optimizations. With a small uncertainty value, the heuristic strategy is close to our adaptive strategy, which re-optimizes at estimation errors of one tuple. In our experiments, our adaptive execution strategy is superior to the heuristic execution strategy, because each re-optimization improves the plan choice with true information. In almost all cases, this improves the plan quality. In addition, our adaptive execution strategy is independent of constants, such as the uncertainty value.

5.5.2 *Execution Time*

Section 5.5.1 reveals that our adaptive execution strategy identifies plans with significantly improved true costs compared to the initial or estimated optimal plans. Ideally, better true costs also result in faster end-to-end query execution times. There are two additional pre-conditions to achieve query execution time improvements. First, the cost model has to be accurate so that better true costs result in the faster execution of plans. This is in general essential for each query optimizer. Second, the re-optimization and adaption effort has to be small enough to not entirely consume the execution time gains of the cheaper plan. Since our cost model is not entirely accurate, there are cases in our experiments where we observe a degradation of query execution time, although the true costs are improved and the re-optimization effort is low. In the following experiments, we compare the end-to-end query execution times of our adaptive execution AE against the heuristic execution HE, the conventional execution of the estimated optimal plan EO, and the fastest plans FA.

Figure 5.13 illustrates some typical results for the Join Order Benchmark. We choose the nine worst and the nine best queries with respect to the speedup of end-to-end query execution time of the adaptive execution AE relative to the conventional execution EO. We show the same queries and in the same order as in Figure 5.11. The queries are plotted along the x-axis, and ordered by increasing speedup of the adaptive execution AE. On the y-axis, we show the median end-to-end query execution time over 101 executions in logarithmic scale. The end-to-end query execution time consists of the initial optimization time, the actual plan execution time, and for AE and HE the adaption effort. The adaption effort contains the Steps 4 to 7 of our adaptive execution strategy in Figure 5.4. Since the logarithmic scale aggravates a comparison between initial optimization time, adaption effort and plan execution time, we do an additional comparison in Section 5.5.4. The fastest plans have no optimization time because they are identified in an offline analysis that executes the 500 estimated cheapest plans, which can take several hours per query. Therefore, we only illustrate the pure plan execution time of the fastest plans FA. The fastest plans can help to identify whether further im-

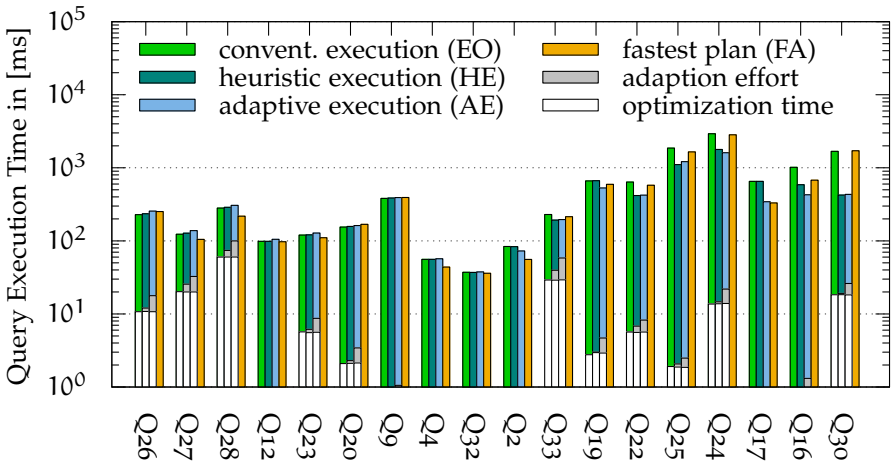


Figure 5.13: Typical results of experiment comparing the end-to-end query execution times on Join Order Benchmark queries.

provements are possible. For EO, HE and AE, the initial optimization time of one query should be identical.

The results of the Join Order Benchmark in Figure 5.13 reveal that the heuristic execution HE and the adaptive execution AE achieve in some queries, such as Q22 and Q30 a similar improvement of end-to-end query execution time. But there are some queries where the adaptive execution AE achieves significantly better results compared to the heuristic execution HE, e.g., Q19, Q17, and Q16. In particular interesting is that the heuristic execution HE and adaptive execution AE can be considerably faster compared to the fastest plans FA, e.g., in Q25, Q24, and Q30. The root cause is that each fastest plan is selected from the 500 estimated cheapest plans, but there are even faster plans beyond 500 estimated cheapest plans, which are identified by HE and AE. The adaptive execution AE has its best speedup at Q30 with $+3.88\times$, and worst regression at Q26 with $-1.12\times$. Since the adaptive execution AE has no true cost degradations in Table 5.1, the query execution time degradations are caused by cost model inaccuracies and adaption effort. The heuristic execution HE has its best speedup also at Q30 with $+3.96\times$, and worst regression also at Q26 with $-1.03\times$. Nevertheless, the adaptive execution AE outperforms the heuristic execution HE with an average speedup relative to conventional execution of $+1.47\times$ compared to $+1.38\times$.

Figure 5.14 shows some typical results for the synthetic benchmark queries with random topology. The queries, plotted along the x-axis are the same and in the same order as in Figure 5.12. Included is Q26, where the adaptive execution AE achieves its best speedup of $+5.19\times$ compared to conventional execution EO, and Q64 where AE has its worst regression of $-1.39\times$. The root cause for the query execution time degradation in Q64 can be observed in Figure 5.12. In Fig-

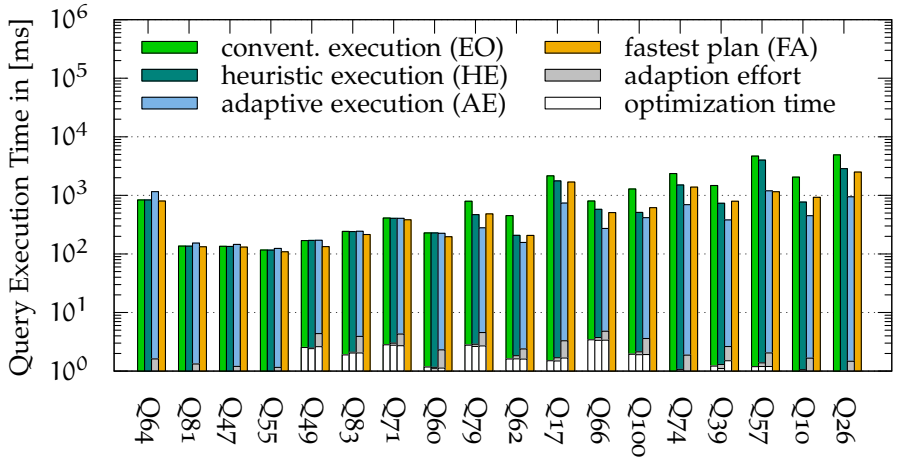
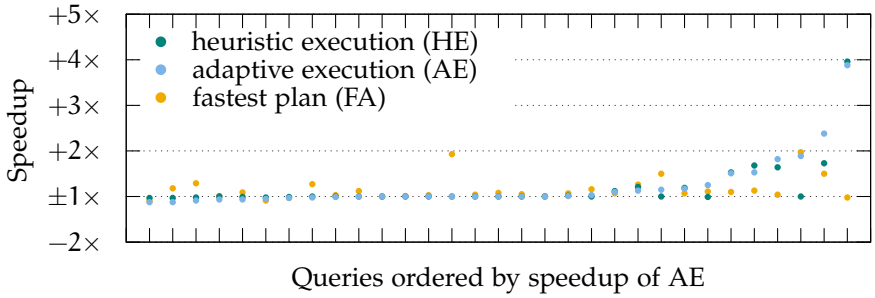


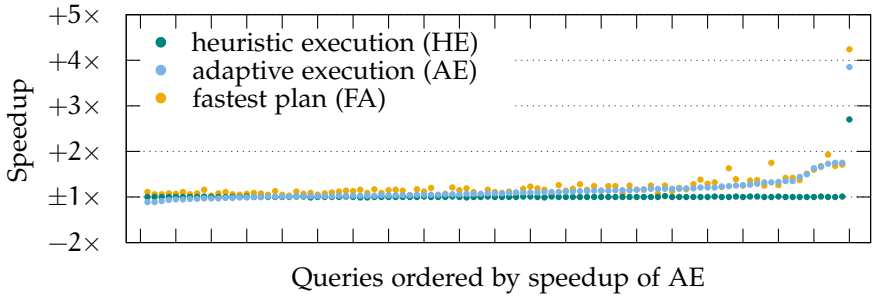
Figure 5.14: Typical results of experiment comparing the end-to-end query execution times on synthetic queries with random topology.

ure 5.12, Q64 has a true cost degradation, which is the consequence of cardinality estimation error during the re-optimization. The remaining degradations of the adaptive execution AE are caused by cost model inaccuracies and adaption effort. The heuristic execution HE achieves its best speedup at Q10 with $+2.67\times$, and has only minor regressions of at most $-1.01\times$, which are measuring deviations. For the majority of queries, the adaptive execution achieves considerably better query execution times compared to conventional execution EO and heuristic execution HE. There are again cases where the adaptive execution AE achieves significantly better query execution times compared to the fastest plans FA, because AE executes plans, which are beyond the 500 estimated cheapest plans. In the end, the average speedup relative to the conventional execution EO is dominated by the adaptive execution AE with $+1.87\times$, compared to the heuristic execution HE with $+1.29\times$ and the fastest plans FA with $+1.46\times$.

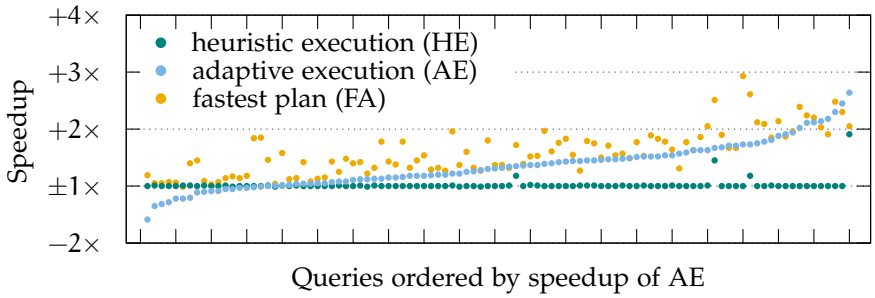
In addition to the typical results, Figure 5.15 and 5.16 illustrate the speedup of heuristic execution HE, adaptive execution AE, and the fastest plans FA relative to conventional execution EO for all queries in all benchmarks. Figure 5.15 illustrates the results for the Join Order Benchmark, and the synthetic benchmark queries with chain and cycle topology. Figure 5.16 contains the synthetic benchmark queries with star, snowflake, and random topology. The queries are plotted along the x-axes and ordered by the speedup of adaptive execution AE. The y-axes show the speedups of end-to-end query execution time. Throughout the benchmark, the adaptive execution AE creates more and stronger speedups than regressions. The Figures 5.15 and 5.16 also reveal that the adaptive execution AE clearly outperforms the heuristic execution HE, which has no improvement in the majority of queries, especially in the synthetic benchmarks. Another



(a) JOB



(b) Chain

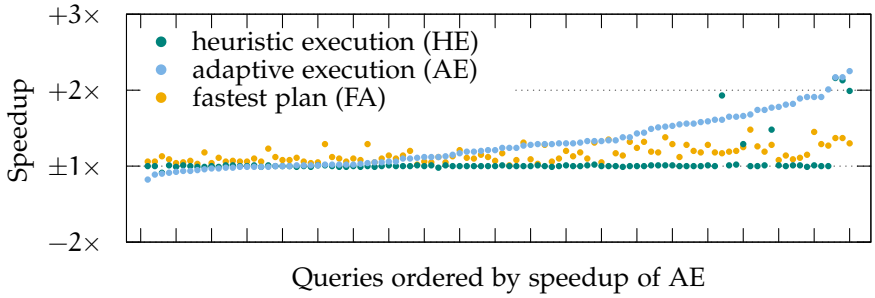


(c) Cycle

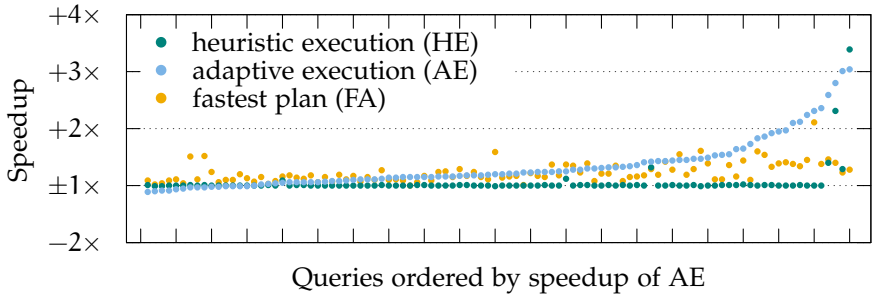
Figure 5.15: Comparison of end-to-end query execution time speedups of heuristic execution (HE), adaptive execution (AE), and fastest plans (FA), relative to the conventional execution of the estimated optimal plan (EO), over all queries in the Join Order Benchmark, and the synthetic chain and cycle benchmarks.

observation is that the adaptive execution outperforms the fastest plans in the synthetic benchmarks with a complex query topology, i.e., star, snowflake, and random. The reason is that complex query topologies have a considerably larger plan space compared to simple topologies, such as chain and cycle. Therefore, the probability that the 500 estimated cheapest plans, which we execute to identify the fastest plan FA, contain a truly fast plan is lower for the complex query topologies compared to the simple query topologies.

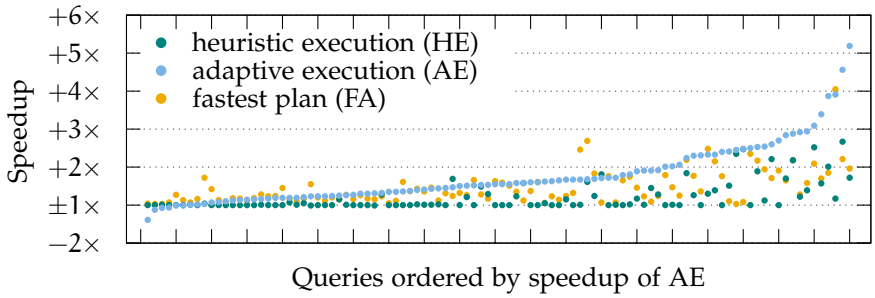
We summarize our experimental results for all benchmarks in Table 5.2, through the accumulated query execution time, the best speed-



(a) Star



(b) Snowflake



(c) Random

Figure 5.16: Comparison of end-to-end query execution time speedups of heuristic execution (HE), adaptive execution (AE), and fastest plans (FA), relative to the conventional execution of the estimated optimal plan (EO), over all queries in the synthetic benchmarks with star, snowflake, and random query topology.

up, the worst regression, and the average of speedups and regressions over all queries in the corresponding benchmark. The adaptive execution AE achieves the best speedup in the star and random queries with $+2.25\times$ and $+5.19\times$, the heuristic execution HE in the Join Order Benchmark and the snowflake queries with $+3.96\times$ and $+3.39\times$, and the fastest plans in the chain and cycle queries with $+4.23\times$ and $+2.94\times$. Both, heuristic execution HE and adaptive execution AE have a stronger best speedup than worst regression in all benchmarks. The adaptive execution AE has in all benchmarks a stronger worst regression compared to the heuristic execution HE. Nevertheless, AE out-

		Σ time	best speedup	worst regression	average
JOB	EO	13941 ms	–	–	–
	HE	10068 ms	+3.96×	−1.03×	+1.38×
	AE	9494 ms	+3.88×	−1.12×	+1.47×
	FA	12483 ms	+1.98×	–	+1.12×
Chain	EO	18780 ms	–	–	–
	HE	17666 ms	+2.70×	−1.01×	+1.06×
	AE	15539 ms	+3.85×	−1.11×	+1.21×
	FA	14562 ms	+4.23×	–	+1.29×
Cycle	EO	41072 ms	–	–	–
	HE	40373 ms	+1.91×	−1.01×	+1.01×
	AE	31100 ms	+2.64×	−1.59×	+1.32×
	FA	25539 ms	+2.94×	–	+1.61×
Star	EO	178116 ms	–	–	–
	HE	177231 ms	+1.16×	−1.08×	+1.01×
	AE	174444 ms	+2.25×	−1.18×	+1.02×
	FA	161987 ms	+1.47×	–	+1.10×
Snowflake	EO	53745 ms	–	–	–
	HE	49442 ms	+3.39×	−1.01×	+1.08×
	AE	43307 ms	+3.04×	−1.11×	+1.24×
	FA	44843 ms	+2.11×	–	+1.20×
Random	EO	81924 ms	–	–	–
	HE	63393 ms	+2.67×	−1.01×	+1.29×
	AE	43739 ms	+5.19×	−1.39×	+1.87×
	FA	56273 ms	+4.07×	–	+1.46×

Table 5.2: Overview of experiments on the end-to-end query execution times on different benchmarks, comparing the adaptive execution (AE) against the conventional execution (EO), the heuristic execution (HE), and fastest plans (FA).

performs HE in all benchmarks with a better average speedup. The adaptive execution AE has the overall best average speedup of +1.87 in the synthetic benchmark with a random query topology.

In the end, our experiments on query execution time reveal that our adaptive execution strategy AE converts the true cost improvements we observe in Section 5.5.1 into the best average end-to-end query execution time improvements. Comparing the cases with execution time degradations to the results of our true cost study in Section 5.5.1

reveals that the root causes for query execution time degradations are mostly cost model inaccuracies. In our experiments, we observe that adaptions cause only minor degradations in case the initial or estimated optimal plan EO could not be considerably improved.

5.5.3 *Re-Optimizations and Plan Switches*

Next, we study the number of re-optimizations and plan switches of the heuristic execution strategy HE and our adaptive execution strategy AE. Our goal is to reveal the peak number of plan switches and re-optimizations, and demonstrate that the majority of re-optimizations of our adaptive execution strategy AE result in improved plans. The number of re-optimizations and plan switches is furthermore interesting for our study on adaption effort in Section 5.5.4.

Figure 5.17 illustrates some typical results of the Join Order Benchmark. We show the same queries, and in the same order as in the Figures 5.11, and 5.13. The queries are plotted along the x-axis, and ordered by increasing query execution time speedup of the adaptive execution AE. On the y-axes, we denote the number of re-optimizations and plan switches. Figure 5.17 shows the results of our adaptive execution strategy AE in the upper plot, and the results of the heuristic execution strategy HE in the lower plot.

We observe from Figure 5.17 that in both execution strategies, the majority of re-optimizations cause a plan switch, i.e., a theoretically better plan. Comparing the results of adaptive execution AE with heuristic execution HE reveals that AE has a considerably larger number of re-optimizations. Although the heuristic execution HE has a smaller number of re-optimizations, not all re-optimizations find a better plan. This confirms the limitations of heuristically defined optimality ranges, we reveal in Section 3.3.1. The adaptive execution AE has its peak number of re-optimizations at Q33. All 7 re-optimizations in Q33 find a better plan. The heuristic execution HE has its peak also at Q33 with 2 re-optimizations and 2 plan switches. Comparing the results of Figure 5.17 to the results of Figure 5.11 and Figure 5.13 reveals that a large number of plan switches does not necessarily result in the strongest improvements of true cost and query execution time.

Figure 5.18 illustrates some typical results of the synthetic benchmark queries with a random topology. We show the same queries and in the same order as in the Figures 5.12 and 5.14. The study in Figure 5.18 confirms the results of Figure 5.17. For both strategies, the majority of re-optimizations find a better plan, and the adaptive execution AE has significantly more re-optimizations and plan switches, compared to the heuristic execution HE. In the synthetic benchmark with random query topology, the adaptive execution AE has at most 7 re-optimizations and 7 plan switches. The heuristic execution HE has at most 1 re-optimization and 1 plan switch. Comparing the number

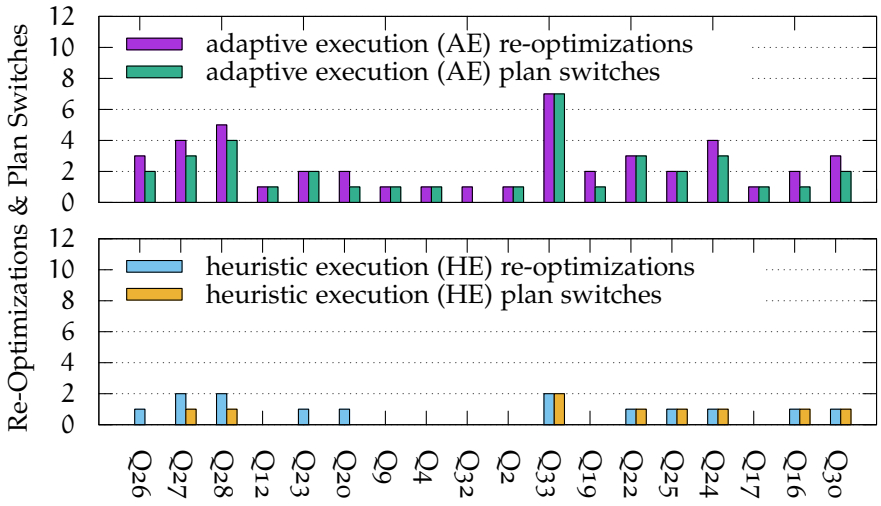


Figure 5.17: Typical results for the number of re-optimizations and plan switches of the adaptive execution (AE) and the heuristic execution (HE) on the Join Order Benchmark.

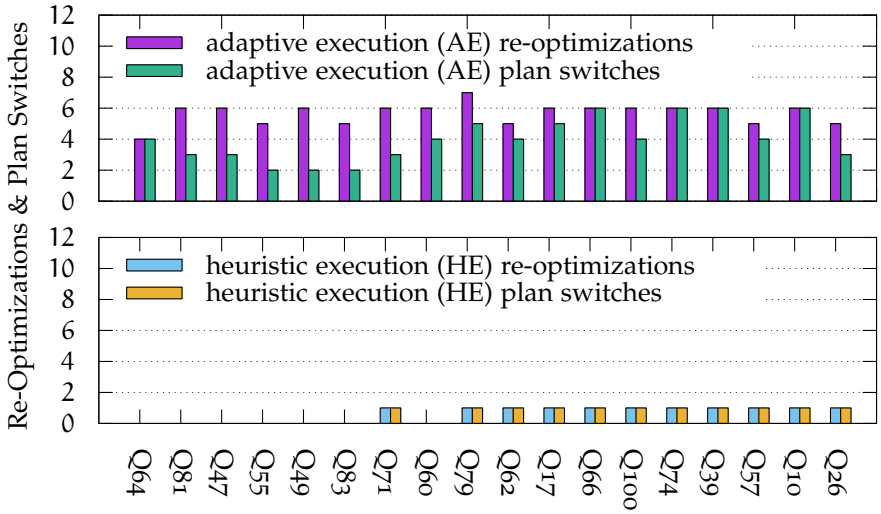


Figure 5.18: Typical results for the number of re-optimizations and plan switches of the adaptive execution (AE) and the heuristic execution (HE) on the synthetic queries with random topology.

of plan switches in Figure 5.18 with the true cost improvements in Figure 5.12, and the query execution time improvements in Figure 5.14, confirms that the number of plan switches does not necessarily correlate with the improvements of true cost and query execution time.

We summarize the study of re-optimizations and plan switches for all benchmarks in the Figures 5.19 to 5.24. For each benchmark, there is one heat-map for the adaptive execution strategy AE and one for the heuristic execution strategy HE. In each heat-map, the

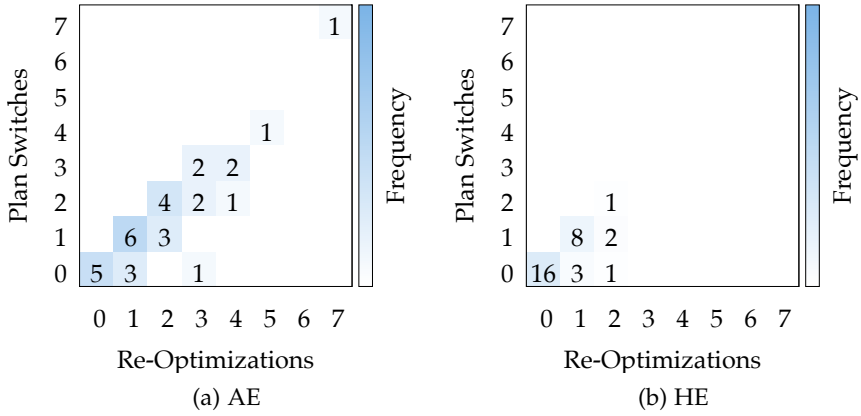


Figure 5.19: Heat map illustrating the number of re-optimizations and plan switches of adaptive execution (AE) and heuristic execution (HE) for all queries in the Job Order Benchmark.

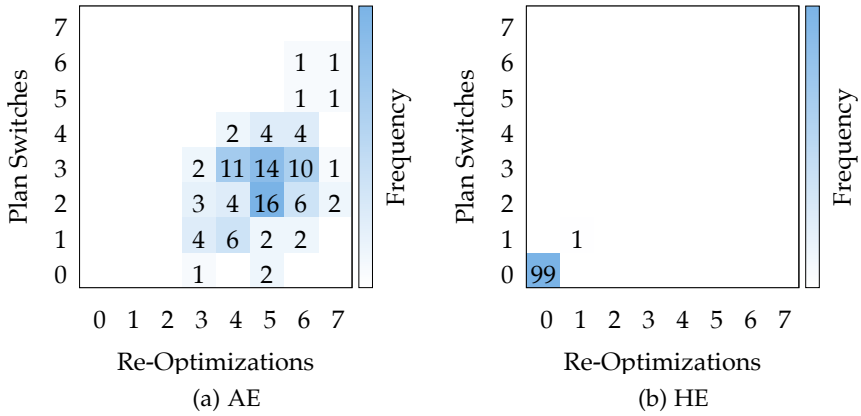


Figure 5.20: Heat map illustrating the number of re-optimizations and plan switches of adaptive execution (AE) and heuristic execution (HE) for all synthetic queries in with a chain topology.

number of re-optimizations are denoted on the x-axis, and the number of plan switches on the y-axis. The entries in the heat maps are the number of queries, which have the corresponding number of re-optimizations and plan switches. Figure 5.20a for instance reveals that for the synthetic queries with a chain topology, the adaptive execution AE has 11 queries where 4 re-optimizations cause 3 plan switches. The heat maps also reveal the peak number of re-optimizations and plan switches of both execution strategies in all benchmarks.

Ideally, all entries in the heat maps should be on a diagonal starting from 0 re-optimizations and 0 plan switches. In case all entries would be on that diagonal, we can conclude that all re-optimizations result in a plan switch. To confirm that the majority of re-optimizations cause a plan switch, it is sufficient that a large number of entries is

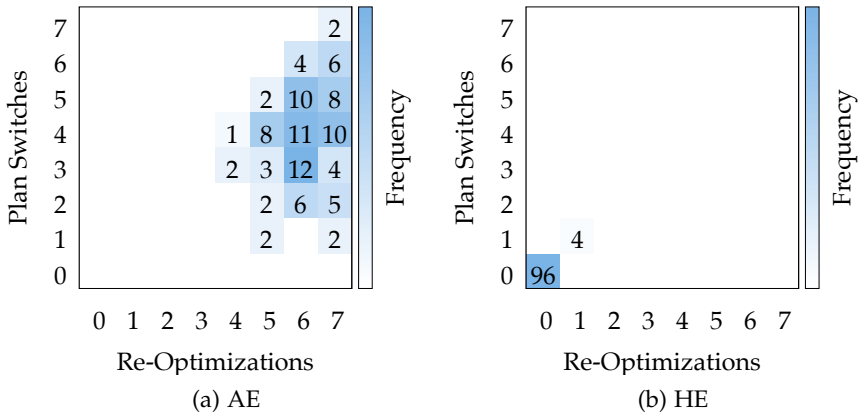


Figure 5.21: Heat map illustrating the number of re-optimizations and plan switches of adaptive execution (AE) and heuristic execution (HE) for all synthetic queries in with a cycle topology.

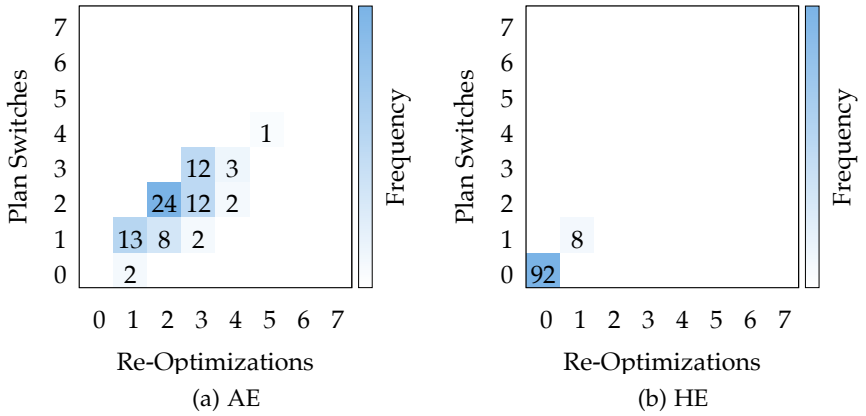
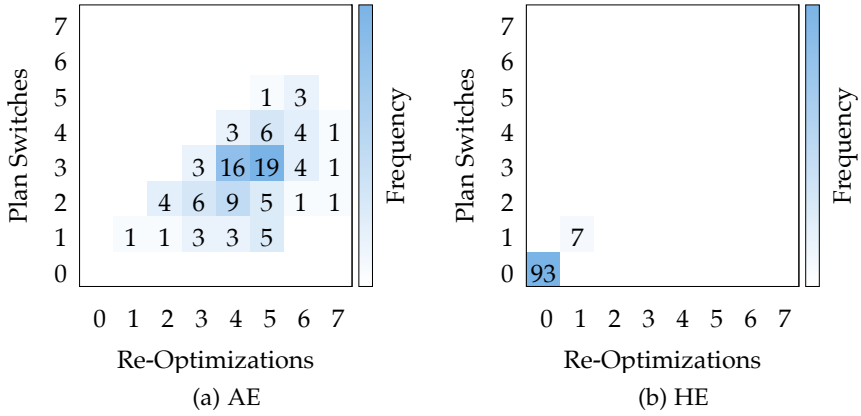


Figure 5.22: Heat map illustrating the number of re-optimizations and plan switches of adaptive execution (AE) and heuristic execution (HE) for all synthetic queries in with a star topology.

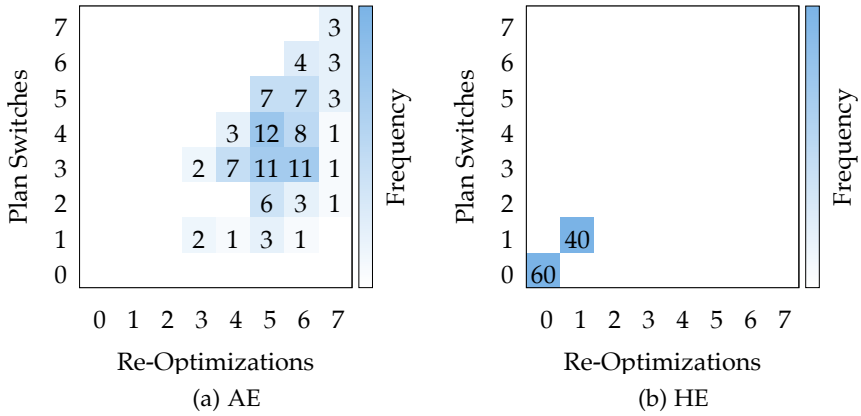
close to the diagonal. The Figures 5.19b to 5.24b reveal that the heuristic execution HE achieves this property in all benchmarks. The adaptive execution AE achieves the property in the Join Order Benchmark in Figure 5.19a, and in the synthetic benchmarks with star, snowflake, and random query topology in the Figures 5.22a, 5.23a, and 5.24a.

From our study on re-optimizations and plan switches, we conclude that our adaptive execution strategy AE triggers only a small number of re-optimizations, which is in the same order of magnitude as the number of operators. We furthermore observe that the majority of re-optimizations find a better query execution plan. Although the heuristic execution HE does considerably less re-optimizations, our experiments reveal that the majority, but not all of these re-optimizations can find a better query execution plan.



(a) AE (b) HE

Figure 5.23: Heat map illustrating the number of re-optimizations and plan switches of adaptive execution (AE) and heuristic execution (HE) for all synthetic queries in with a snowflake topology.



(a) AE (b) HE

Figure 5.24: Heat map illustrating the number of re-optimizations and plan switches of adaptive execution (AE) and heuristic execution (HE) for all synthetic queries in with a random topology.

5.5.4 *Adaption Effort*

At the end of our experimental evaluation, we study the query execution time consumption of our adaptive execution strategy AE, and compare it with the heuristic execution strategy HE. For both execution strategies, the adaption effort consists of the execution time for the Steps 4 to 7 in Figure 5.4, accumulated over all pipelines in the query execution plan. The execution time for the Steps 4, 5, and 7 are always below 1.00 ms, so that the adaption effort mostly quantifies the effort for the selective re-enumerations. Figure 5.25 illustrates some typical results for the Join Order Benchmark. We show the same queries and in the same order as in the Figures 5.11, 5.13, and 5.17. The queries are plotted along the x-axis and ordered by the end-to-

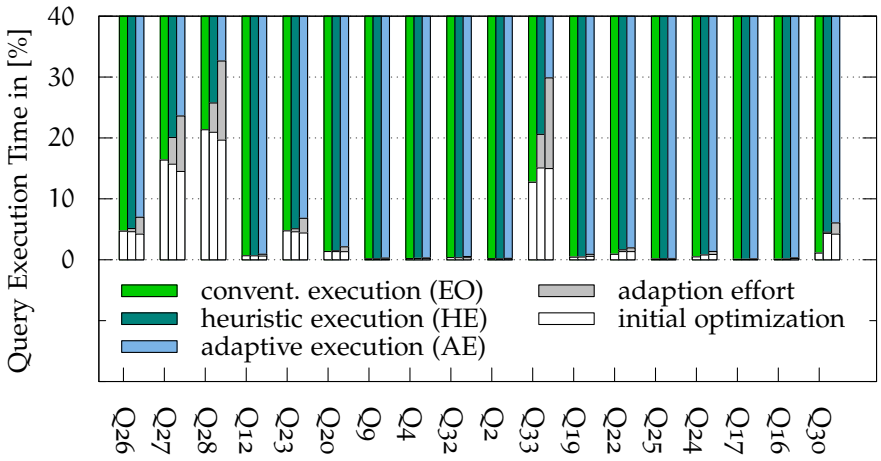


Figure 5.25: Typical results for the adaption effort and initial optimization time of our adaptive execution (AE) compared to conventional execution (EO), and heuristic execution (HE) on queries of the Join Order Benchmark.

end query execution time speedup of our adaptive execution AE. In contrast to Figure 5.13, which shows the end-to-end query execution times in logarithmic scale on the y-axis, Figure 5.25 shows the query execution time percentages of initial optimization and adaption effort on the y-axis. We compare our adaptive execution AE against heuristic execution HE, and furthermore show the initial optimization time of conventional execution EO. Since there is no logarithmic scale, the bars of initial optimization time and adaption effort in Figure 5.25 can be directly compared. Note that the initial optimization times of EO, HE and AE are identical in absolute numbers.

In the Join Order Benchmark, there are no queries where the adaption effort of AE exceeds the initial optimization time. We observe the peak adaption effort of AE in Q33: According to Figure 5.13, the end-to-end query execution time of AE in Q33 is 195.43 ms. The 7 re-optimizations, illustrated in Figure 5.17, result in an adaption effort of 29.09 ms, i.e., 14,89%, compared to 29.27 ms, i.e., 15.00% for the initial optimization time. Despite this considerable adaption effort, the adaptive execution AE of Q33 results in a query execution time speedup of $+1.17\times$. According to Figure 5.13 the conventional execution EO takes 229.41 ms compared to 195.43 ms for the adaptive execution AE. Like the initial optimization, the adaptive execution AE is single threaded. Consequently, the adaptive execution AE does not only improve the end-to-end query execution time, but also increases the percentage of single threaded execution time, which has an additional positive impact on the consumed CPU time.

Our experiments also reveal that the adaption effort mostly depends on the size of the not-yet-executed query parts, and the query

query graph topology. Comparing Q33 with Q24 in the Figures 5.17 and 5.25 reveals that the number of re-optimizations only has a minor impact on the adaption effort of AE. While Q33 has 7 re-optimizations, as illustrated in Figure 5.17, and the large adaption effort according to Figure 5.25, Q24 has 4 re-optimizations and a marginal adaption effort. Due to the design of our adaptive execution engine, there is no execution time difference between a plan switch and no plan switch. The overall adaption effort mostly results from the effort for selective re-enumerations. We observe that, although we perform multiple selective re-enumerations during a query execution, the overall adaption effort is in the same order of magnitude as an initial optimization. The reason is that we only re-enumerate the not-yet-executed query parts, and the enumeration effort can grow exponentially with the query size, i.e., the number of referenced tables and intermediate results. Consequently, the smaller the query size, the smaller the enumeration effort.

We also observe queries in the Join Order Benchmark, where the adaptive execution AE introduces slight degradations of end-to-end query execution time. In Q28 for instance, the adaptive execution AE does 5 re-optimizations and 4 plan switches, according to Figure 5.17. The plan switches reduce the pure plan execution time plus optimization time to 266.43 ms. The conventional execution EO in contrast needs 281.09 ms, as illustrated in Figure 5.13. Since the adaptive execution AE introduces an effort of 39.79 ms, the end-to-end query execution time of AE is 306.22 ms, which eventually is a degradation of $-1.08\times$ compared to EO. On average the adaptive execution AE uses 1.27% of the query execution time for adaptions, and 1.91% for initial optimizations. For the queries in Figure 5.25, the heuristic execution HE has smaller adaption effort. On average, the heuristic execution HE spends 0.36% of the query execution time for adaptions, and 1.80% for initial optimizations. According to Table 5.2, the smaller adaption effort of the heuristic execution HE also results in a smaller average improvement of end-to-end query execution time.

Figure 5.26 illustrates some typical results for the synthetic queries with random topology. We show the same queries and in the same order as in the Figures 5.12, 5.14, and 5.18. Figure 5.26 basically confirms the results of Figure 5.25. Compared to the Join Order Benchmark results, the peak optimization time and the adaption effort are significantly smaller, because all synthetic queries join at most 10 tables. In the synthetic queries with random topology, there are 73 of 100 queries where the adaption effort exceeds the initial optimization time. Nevertheless, we observe that there are no queries where the adaption effort considerably exceeds the initial optimization time.

Figure 5.16c reveals that in 5 of the 100 queries with random topology, the adaptive execution AE causes a degradation of end-to-end query execution time. Only in Q49, the adaption effort is the root

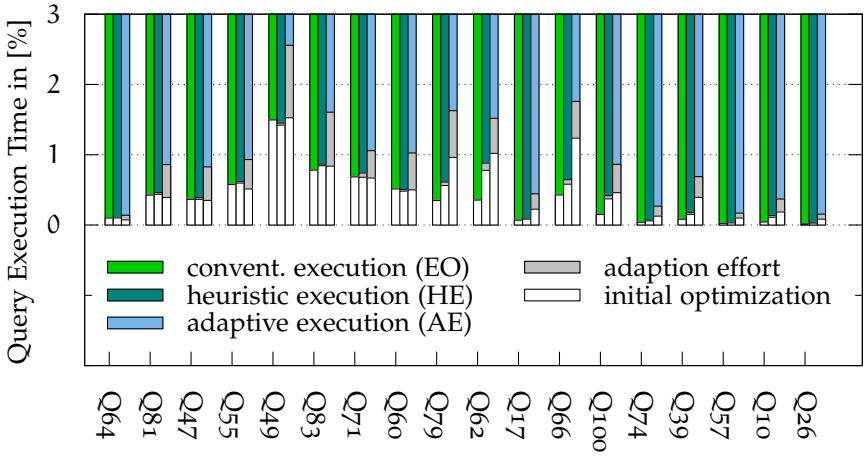


Figure 5.26: Typical results for the adaption effort and initial optimization time of our adaptive execution (AE) compared to conventional execution (EO), and heuristic execution (HE) on synthetic benchmark queries with random topology.

cause for a degradation of end-to-end query execution time ($-1.01\times$). The adaption effort in the remaining 4 queries is below 1 ms. According to Figure 5.12, the degradation in Q64 results from increased true cost of the adaptive plan AE compared to estimated optimal plan EO. For Q81, Q47, and Q55, Figure 5.12 shows improved true costs of the adaptive plans AE compared to the estimated optimal plans EO. Since the adaption effort in Q81, Q47, and Q55 is below 1 ms, we conclude that the corresponding query execution time degradations of at most $-1.12\times$ result from inaccuracies in the cost model.

We summarize the end-to-end query execution time percentages of initial optimization and adaptions for all benchmarks in Table 5.3. We denote the initial optimization time as t_o, and summarize it by maximum, minimum, and average t_o in percent. Similarly, we denote the overall adaption time of a query as t_a, and summarize it by maximum, minimum, and average t_a in percent. Table 5.3 reveals that the peak adaption effort for our adaptive execution AE is 14,89%. Despite the adaption time of 14,89%, the end-to-end query execution time of the corresponding Join Order Benchmark query is improved by $1.17\times$. According to Table 5.3, the average adaption time for our adaptive execution AE is at most 1.27%. Furthermore, we observe that the execution times of adaptions of AE are similar to the execution time of the initial optimizations in all benchmarks. Table 5.3 further reveals that the peak adaption percentage correlates with the peak initial optimization percentage in all benchmarks. This confirms that the adaption effort mostly depends on the query complexity, i.e., the number of referenced tables and the query graph topology.

To conclude our findings, the adaptive execution AE spends only a small amount of execution time for adaptions, which are in the

		max t_o	min t_o	avg t_o	max t_a	min t_a	avg t_a
JOB	EO	21.34%	0.06%	1.30%	–	–	–
	HE	20.93%	0.06%	1.80%	5.51%	0.00%	0.36%
	AE	19.64%	0.07%	1.91%	14.89%	0.00%	1.27%
Chain	EO	0.31%	0.02%	0.17%	–	–	–
	HE	0.31%	0.05%	0.18%	0.03%	0.01%	0.02%
	AE	0.33%	0.07%	0.22%	0.44%	0.07%	0.27%
Cycle	EO	0.55%	0.05%	0.16%	–	–	–
	HE	0.56%	0.05%	0.16%	0.04%	0.00%	0.01%
	AE	0.57%	0.05%	0.21%	0.82%	0.06%	0.31%
Star	EO	2.78%	0.03%	0.17%	–	–	–
	HE	2.72%	0.03%	0.17%	0.24%	0.00%	0.00%
	AE	3.04%	0.03%	0.18%	1.53%	0.01%	0.08%
Snowflake	EO	0.73%	0.01%	0.16%	–	–	–
	HE	0.71%	0.02%	0.17%	0.06%	0.00%	0.01%
	AE	0.81%	0.02%	0.21%	0.90%	0.01%	0.19%
Random	EO	1.50%	0.02%	0.17%	–	–	–
	HE	2.14%	0.03%	0.22%	0.14%	0.00%	0.02%
	AE	2.07%	0.04%	0.33%	1.77%	0.04%	0.28%

Table 5.3: Overview of adaption effort and initial optimization time of conventional execution (EO), heuristic execution (HE), and adaptive execution (AE).

same order of magnitude as the initial optimizations, and in multiple queries even smaller. The reason is that the adaption effort mostly depends on the overall re-enumeration effort, which becomes smaller with each re-enumeration, because we re-optimize only the not-yet-executed query parts. We observe that the heuristic execution HE has even smaller adaption efforts, but also higher end-to-end query execution times, as illustrated in Table 5.2. The reason is that the heuristic execution HE misses a majority of better plans, because of its conservative re-optimization strategy. In our experiments, we only find a marginal number of queries, where the effort of our adaptive execution strategy AE causes a degradation of query execution time, and those degradations are negligible. Even if the end-to-end query execution time is not improved by our adaptive execution strategy AE, the adaption effort is single threaded and therefore consumes less CPU time compared to the multi-threaded pipeline execution.

5.6 RELATED WORK

A comprehensive overview of related work on Mid-Query Re-Optimization, and robust and adaptive query processing can be found in the surveys by Deshpande et al. [47] and Yin et al. [80]. To compare the different robust and adaptive query processing approaches, Yin et al. [80] describe a set of criteria, which are: (1) the addressed estimation error source, e.g., non-uniform data distribution, missing statistics, or data modifications, (2) the target query type, e.g., the currently running query, a future query, or a parametric query, (3) the target optimization decision, e.g., operator order, operator implementation, access method, or resource allocation, (4) the risk of performance degradations, and (5) the engineering cost. The comparison by Yin et al. [80] does not consider that some approaches that address parametric queries require considerable offline effort to analyze the plan space before they can make a decision. Therefore, those approaches are not applicable at query optimization time. We start discussing related work on Mid-Query Re-Optimization, i.e., approaches to improve the currently running query with runtime feedback.

With *Dynamic Re-Optimization*, Kabra and DeWitt [31] initially described Mid-Query Re-Optimization. Dynamic Re-Optimization always re-optimizes in pipeline breakers on complete intermediate results, and never adds artificial pipeline breakers. This is identical to our adaptive execution strategy. Kabra and DeWitt [31] present the heuristically defined optimality ranges as re-optimization criterion, which we discuss in Section 3.4. According to Kabra and Dewitt [31] an optimality range can be defined as follows: Based on an uncertainty value U and an estimate E, the lower bound of an optimality range for cardinality $f^{\downarrow} = E - (E * 0.1 * U)$ and the upper bound $f^{\uparrow} = E + (E * 0.2 * U)$. Kabra and Dewitt [31] furthermore present a set of rules to derive if the uncertainty value U of an estimate is low, medium or high, but do not specify actual values for U. We illustrate in Section 3.3.1 that heuristics cannot characterize optimality ranges precisely. In the experiments in Section 5.5, we denote the approach by Kabra and Dewitt [31] as heuristic execution HE. Section 5.5.3 reveals that the uncertainty value U is a weak point. To achieve comparable results to our adaptive query processor, the uncertainty value U has to be adjusted for each workload. We do not rely on constants at all, and re-optimize at an estimation error of one or more tuples, which guarantees to not miss better plans.

Progressive Optimization by Markl et al. [39] is a valuable improvement on Dynamic Re-Optimization with better optimality ranges and additional re-optimization points. As we explain in Section 3.4, the optimality ranges in Progressive Optimization are denoted as validity ranges. Validity ranges are calculated through the comparison with structurally equivalent plans, which have the same operator order

but different operator implementations and different build and probe sides of hash join operators. Consequently, only a subset of the necessary plan alternatives is considered, and the validity ranges are not precise. Nevertheless, a re-optimization triggered through a validity range is always correct. In case a true cardinality is outside of the validity range, there is a better plan. In contrast, it cannot be guaranteed that there is no better plan when the cardinality is inside of the validity range. Therefore, switches to better plan alternatives can be missed. In contrast to Progressive Optimization, our work does not miss better plans, because it re-optimizes at cardinality estimation errors of one or more tuples. Markl et al. [39] furthermore describe additional re-optimization points, denoted as checkpoints, and where they can be placed in a query execution plan. The *materialization checkpoint* is an artificial pipeline breaker, which increases the re-optimization opportunities but at the same time also the risk of query execution time degradations. *Eager checkpoints* interrupt the pipeline if an intermediate result exceeds the upper bound of a validity range, and therefore cause incomplete intermediate results. The incomplete intermediate results cannot be re-used, and are wasted. In addition, re-optimizing on incomplete intermediate results, i.e., with wrong cardinalities, can cause a wrong plan choice.

Proactive Re-Optimization by Babu et al. [41] is built on Dynamic Re-Optimization [31]. It uses the same heuristics to define an optimality range around a cardinality estimate, and triggers a re-optimization when a true cardinality is outside the heuristic optimality range. Within the heuristically defined optimality range, Proactive Re-Optimization tries to identify either a single optimal plan, a switchable plan, or a robust plan. Therefore, we discuss Proactive Re-Optimization also in Section 4.7. Proactive Re-Optimization searches the optimal plan for the estimated cardinality and the cardinality at the lower and upper bound. In case all three plans are identical, there is a single optimal plan. If one of the three plans has estimated cost, which are at most a certain threshold larger than the corresponding optimal plan at that cardinality, Proactive Re-Optimization identifies a robust plan. In case there is neither a single optimal plan, nor a robust plan, Proactive Re-Optimization tries to identify a switchable plan, which requires that all three plan alternatives are structurally equivalent, as in Progressive Optimization [39]. When neither a single optimal, nor robust, nor switchable plan can be identified, Proactive Re-Optimization choses the optimal plan at the estimated cardinality, and assumes that it is optimal within the heuristically defined optimality range, although it could not identify a single optimal plan. Proactive Re-Optimization is an improvement on Dynamic Re-Optimization, but it is still based on heuristically defined optimality ranges. Furthermore, it considers at most, two additional plan alternatives per re-optimization point and heuristic optimality

range, although the overall plan space can grow exponentially with the number of referenced tables. At query execution time, Proactive Re-Optimization monitors the cardinalities of incomplete intermediate results to make decisions on plan switches and re-optimizations, which can lead to wrong optimization choices.

Incremental Execution by Neumann and Galindo-Legaria [69] is an interesting orthogonal improvement on Mid-Query Re-Optimization. Like the majority of Mid-Query Re-Optimization approaches, Incremental Execution starts with the estimated optimal plan. The core of Incremental Execution is an algorithm to identify sub-plans, whose intermediate results have a strong impact on the overall plan choice. Those crucial sub-plans are executed first. After each execution, the optimizer is invoked again with the true cardinalities of the crucial sub-plans. To be applicable in practice, the core algorithm of incremental execution only considers linear join trees, and no bushy join trees. In contrast to our work, Incremental Execution adds artificial pipeline breakers as re-optimization points, which can considerably increase the query execution time. As we do, Incremental Execution re-optimizes only on complete intermediate results. We discuss in Section 7.1.3 how the work by Neumann and Galindo-Legaria [69] can improve the pipeline selection in our adaptive query processor.

Incremental Re-Optimization by Liu et al. [84] is another orthogonal improvement on Mid-Query Re-Optimization. Incremental Re-Optimization originates from stream databases, and has the goal to speedup re-optimizations. The transformation based query optimizer of Incremental Re-Optimization is implemented in `datalog` [52], and has a state, which is consecutively updated. Because of the implementation in `datalog`, each cardinality update is propagated to all previously enumerated plans, so that their costs are updated, and a new optimal plan may be identified. Liu et al. [84] furthermore describe how previously pruned plans can be re-enumerated, to guarantee a correct re-optimization result. Compared to Liu et al. [84], our join optimizer and selective re-enumeration algorithm do dynamic programming optimization and are implemented in `C++`. We also keep the optimizer state, i.e., the plan table of the initial plan search. The plan table of our query optimizer is a conventional data structure allocated in memory. We can arbitrarily modify the plan table, e.g., to reference intermediate results as optimal plan of a plan class, or set new optimal plans during a selective re-enumeration. Like Liu et al. [84], we also re-enumerate all necessary plan alternatives, including plans that were previously pruned. In contrast to Liu et al. [84], we can describe the plan space of our selective re-enumeration.

There is a large variety of other Adaptive Query Processing approaches. *Eddies* by Avnur and Hellerstein [32] are tuple routing operators, which can, for instance, be used to improve the order of selection operators. An Eddie routes single tuples to different operators. It

monitors the selectivity of the single operators, to process operators with the highest selectivity first. Consequently, the different tuples in the final result are calculated through different query execution plans. Eddies require bookkeeping for each tuple, to decide which operators are already executed. This is unsuitable for the highly optimized parallel pipelines we explain in Section 2.5.2. Furthermore, Eddies make optimization choices based on incomplete intermediate results. Li et al. [49] extend the idea of Eddies for join queries and pipelined execution in batches. They only consider linear join trees, and change the probe order of batches, depending on the join selectivity.

Parametric Query Optimization [37, 38, 55], identifies query execution plan alternatives before the execution starts, so that the plan choice can be made at query execution time based on true statistics without invoking the optimizer. Parametric Query Optimization requires either a comprehensive and expensive analysis of the plan space [37, 38], or multiple query executions and the caching of the executed query execution plans [55]. Related to Parametric Query Optimization are Plan Diagrams [45, 53] and Plan Bouquets [71], which require considerable pre-calculation effort. True cardinalities, derived at query execution time can also improve the quality of future query execution plans. The *Learning Optimizer* (LEO) by Stillger et al. [36] employs a monitoring component to derive statistics such as true cardinalities, and predicate selectivities from executed queries. LEO stores the gathered information in a feedback cache. The information from the feedback cache, together with the database catalog can improve cardinality estimates and therefore the plan quality.

5.7 CONCLUSION

In this chapter, we present an adaptive query processor to compensate cardinality estimation errors and sub-optimal query execution plans with true cardinalities retrieved from materialized intermediate results. Our adaptive query processor combines the findings of the related work on Mid-Query Re-Optimization and our findings of the work on optimality ranges in Chapter 3. To keep the risks for query execution time degradations low, our adaptive query processor re-optimizes only in existing pipeline breakers that contain complete intermediate results. Our adaptive query processor furthermore re-optimizes only the query parts, which have not yet been executed, because this guarantees progress, and steadily reduces the size of the optimization problem. Due to the tight integration of query execution engine and query optimizer, our adaptive query processor can speedup re-optimizations and query execution plan switches. The query execution engine can directly inject intermediate results with true cardinalities into the plan table of the query optimizer. Furthermore, our selective re-enumeration algorithm enumerates only those

query execution plans, which are directly affected by cardinality updates. Through the native support of adaptivity, the actual switch from one query execution plan to another creates no overheads in the query execution engine.

Our adaptive query processor has a re-optimization criterion, which re-optimizes at cardinality estimation errors of one or more tuple. Nevertheless, our experimental evaluation reveals that the overall execution time for re-optimizations and adaptions is, in most queries, similar or even smaller compared to the initial optimization time. The main reason is that we only re-optimize the not-yet-executed query parts, which consecutively decreases the re-enumeration effort. Our experiments also reveal that the majority of re-optimizations identify a better query execution plan. Although competitive approaches have a more conservative re-optimization strategy and do less re-optimizations, we demonstrate that they cannot find better query execution plans with each re-optimization. Our experiments reveal that more re-optimizations, and therefore more true cardinalities cause better end-to-end query execution times. In our experiments, our adaptive query processor improves the query execution times on a real-world database by up to 3.88× and on average by 1.47×. For the synthetic benchmark, our adaptive query processor improves the query execution times by up to 5.19× and on average by 1.87×. We also observe degradations of end-to-end query execution time, but they are smaller and less frequent compared to the improvements.

All in all, our work demonstrates that using true cardinalities to efficiently improve query execution plans with Mid-Query Re-Optimization has a considerable impact on the query processor architecture. Nevertheless, our experiments reveal that true cardinalities are superior to estimated cardinalities. True cardinalities can improve the query execution plan of the currently running query, which can result in significant improvements of end-to-end query execution time.

6 IMPLICATIONS ON QUERY PROCESSING

Our work addresses unsolved problems and limitations of existing work to detect and compensate sub-optimal query execution plans, caused by cardinality estimation errors. In this chapter, we discuss the boundaries between our different contributions and the implications on query processing.

6.1 DETECTING SUB-OPTIMAL QUERY EXECUTION PLANS

While it is trivial to detect cardinality estimation errors by comparing the estimated cardinality with the true cardinality observed at query execution time, it is not trivial to decide if the cardinality estimation error makes another plan optimal. To address this problem, we present in Chapter 3 an algorithm to calculate precise optimality ranges for the cardinality of intermediate results. A true cardinality that is outside of the corresponding optimality range indicates that the optimizer can find another, cheaper plan.

Our work on Mid-Query Re-Optimization in Chapter 5 reveals that optimality range calculation has similarities to our selective re-enumeration algorithm. Both consider a similar set of plans, namely those plans that directly or indirectly reference a certain sub-plan. While optimality range calculation assumes that the cardinality of the referenced sub-plan is variable, selective re-enumeration has a true cardinality value for the referenced sub-plan. According to the experimental results in Figure 5.10, optimality range calculation enumerates at least the number of plans that selective re-enumeration enumerates. Therefore, we conclude that the number of enumerated plans by selective re-enumeration can be a lower bound for the number of enumerated plans by optimality range calculation. We furthermore make the following conclusions from the experimental results in Figure 5.10: When the decision, whether a better query execution plan for a certain cardinality can be found, has to be made once only, such as in mid-query re-optimization, the selective re-enumeration is superior to the calculation of optimality ranges. However, the complexity of optimality ranges can amortize when the decision has to be made multiple times, such as in parametric queries or plan caching.

6.2 COMPENSATING SUB-OPTIMAL QUERY EXECUTION PLANS

A complementary problem to the detection of sub-optimal query execution plans is the compensation of sub-optimal query execution

plans, caused by cardinality estimation errors. Although cardinality estimation errors are evident, conventional query optimizers still select the estimated cheapest query execution plan based on estimated cardinalities. Therefore, we present in Chapter 4 metrics to quantify the robustness of query execution plans with respect to cardinality estimation errors, and a new robust plan selection strategy. In case of cardinality estimation errors, robust plans can process a query faster than the estimated cheapest plans. Another shortcoming of conventional query optimizers is that they make decisions solely based on estimated cardinalities. In Chapter 5, we present our improvements on Mid-Query Re-Optimization. We demonstrate an adaptive query processor that steadily improves the estimated cheapest plan with true cardinalities, which are collected in the query execution engine.

Both, our robust plan selection strategy and our adaptive query processor improve the currently running query. Like our calculation algorithm for optimality ranges, our robust plan selection strategy and the adaptive query processor support arbitrary relational operators and join trees, including bushy trees. Since both approaches share the same goal of improving query execution plans, we compare them with respect to their engineering cost, information source, risk of performance degradation, and end-to-end query execution time.

ENGINEERING COST To integrate our robust plan selection into an existing query processor, only the query optimizer has to be modified. The enumeration algorithm has to be extended to enumerate the k-cheapest plans, so that the robust plan candidates can be derived. Furthermore, at least one of our three robustness metrics has to be implemented. In addition, there are some minor changes, such as selecting the estimated most robust query execution plan instead of the estimated cheapest query execution plan from the robust plan candidates. The query execution engine is not modified.

Our adaptive query processor has more significant differences compared to a conventional query processor. The execution engine has to orchestrate the execution of pipelines and the invocations of the query optimizer. Furthermore, the execution engine has to collect true cardinalities, and be aware of constantly changing query execution plans. Our query optimizer is extended to accept intermediate result injections into its plan table, and perform selective re-enumerations. To conclude, the adaptive query processor requires considerably higher engineering cost compared to our robust plan selection strategy.

INFORMATION SOURCE The robust plan selection is performed at query optimization time and therefore solely based on cardinality estimates. Our adaptive query processor starts with the estimated optimal query execution plan, based on estimated cardinalities, and steadily improves the plan choice with true cardinalities. Both ap-

proaches require that some cardinality estimates are close to their true values, and not entirely arbitrary. Our experiments in Section 5.5 reveal that more runtime feedback, i.e., more true cardinalities, achieve stronger improvements of true cost and end-to-end query execution time. We compare the end-to-end query execution times of our robust plan selection and our adaptive query processor below, and show that our adaptive query processor achieves more and stronger improvements compared to robust plan selection.

RISK OF PERFORMANCE DEGRADATION Our robust plan selection identifies a query execution plan from the robust plan candidates that is less sensitive to cardinality estimation errors. The robust plan candidates are based on the k-cheapest query execution plans. Furthermore, our near-optimal plans requirement guarantees that each robust plan candidate has at most λ-times larger estimated cost, compared to the estimated cheapest plan. Nevertheless, in case of correct cardinality estimates, a robust plan can be slower compared to an estimated cheapest plan. Another risk of end-to-end query execution time degradations can be the procedure to select a robust plan. In our experiments, selecting a robust plan consumes only a small single digit percentage of end-to-end query execution time. Nevertheless, the procedure to select a robust plan can introduce an additional overhead compared to conventional plan selection.

The adaptive query processor can also introduce performance degradations. In case of cardinality estimation errors, we cannot guarantee that the re-optimizations find better query execution plans. In addition, we cannot guarantee that plan switches considerably improve the plan quality. Consequently, our adaptive query processor can introduce execution time overheads because of the re-optimization effort. Nevertheless, our experiments in Section 5.5 reveal that the overall re-optimization effort is similar to an initial optimization. We furthermore observe that there is a small number of queries only, where our adaptive execution strategy creates performance degradations. The majority of those performance degradations are rather caused by cost model inaccuracies than by the effort for re-optimizations.

Similar to the execution time consumption of conventional query optimization, the additional effort for robust plan selection and adaptive execution may not amortize in short running queries. Robust plan selection and adaptive execution can both suffer from cost model inaccuracies, so that a theoretically cheaper or more robust plan does not show the expected behavior in practice.

END-TO-END QUERY EXECUTION TIME Since we choose exactly the same benchmarks and system for the experimental evaluation of robust plan selection and adaptive execution, we can directly compare the end-to-end execution times. To better compare the end-to-

end query execution times of robust plan selection and adaptive execution, we combine the Tables 4.1 and 5.2. We compare the results for the Join Order Benchmark, and the synthetic benchmark queries with chain and cycle topology in Table 6.1. Table 6.2 compares the results for the synthetic benchmark queries with star, snowflake, and random topology. We summarize the results of all benchmarks through the accumulated end-to-end query execution time, the best speedup, and the worst regression over all queries in the corresponding benchmark. We furthermore reveal the average of speedups and regressions. Speedups and regressions are relative to end-to-end query execution times of the estimated optimal plans EO. For each benchmark, we show the estimated optimal plans EO as well as the fastest plans FA, which we identify through the execution of the 500 estimated cheapest plans for each query. The Tables 6.1 and 6.2 also show the robust plan selection using either the cardinality-slope metric FS, the selectivity-slope metric SS, or the cardinality-integral metric FI. For Mid-Query Re-Optimization, we show the heuristic execution strategy HE, and our adaptive execution strategy AE.

The Tables 6.1 and 6.2 reveal that our adaptive execution strategy AE outperforms the robust plan selection strategies FS, SS, and FI in all benchmarks with higher peak speedups and higher average speedups. For the queries of the Join Order Benchmark, our adaptive execution strategy AE achieves a peak speedup of 3.88×, and an average speedup of 1.47×, while the best robust plan selection using the cardinality-integral metric FI achieves a peak speedup of 1.83×, and on average a degradation of −1.03×. For the queries of the synthetic benchmarks, our adaptive execution strategy AE achieves its peak speedup of 5.19× and average speedup of 1.87× in the random query benchmark. The robust plan selection achieves its peak speedup of 3.49× with the cardinality-integral metric FI in the chain query benchmark, while the adaptive execution strategy AE achieves a peak speedup of 3.85× in the same benchmark. The Tables 6.1 and 6.2 also reveal that the adaptive execution strategy AE has less severe degradations than the robust plan selection.

Although the adaptive execution strategy AE performs better than the robust plan selection in the majority of queries, there are a few queries, where the robust plan selection has advantages. Comparing the results of JOB Q2 in Figure 4.10 with Figure 5.15 reveals that all robust plan selection strategies achieve a speedup of approximately 1.47×, while the adaptive execution strategy AE achieves a speedup of 1.15× only. Another example is Q7, where the adaptive execution strategy AE achieves no speedup. The robust plan selection in contrast achieves a speedup of 1.29× with the cardinality-slope metric FS, 1.83× with the selectivity-slope metric SS, and 1.82× with the cardinality-integral metric FI. The root cause for the dominance of robust plan selection in these queries is, that the adaptive execution

		Σ time	best speedup	worst regression	average
JOB	EO	13892 ms	–	–	–
	FS	17696 ms	+1.47×	−2.98×	−1.27×
	SS	14581 ms	+1.83×	−1.52×	−1.05×
	FI	14243 ms	+1.83×	−1.52×	−1.03×
	HE	10068 ms	+3.96×	−1.03×	+1.38×
	AE	9494 ms	+3.88×	−1.12×	+1.47×
	FA	12483 ms	+1.98×	–	+1.12×
Chain	EO	18798 ms	–	–	–
	FS	16091 ms	+3.31×	−1.26×	+1.17×
	SS	17061 ms	+1.79×	−1.36×	+1.10×
	FI	16865 ms	+3.49×	−1.13×	+1.11×
	HE	17666 ms	+2.70×	−1.01×	+1.06×
	AE	15539 ms	+3.85×	−1.11×	+1.21×
	FA	14562 ms	+4.23×	–	+1.29×
Cycle	EO	41084 ms	–	–	–
	FS	34587 ms	+2.43×	−1.21×	+1.19×
	SS	32279 ms	+2.43×	−1.27×	+1.27×
	FI	33193 ms	+2.43×	−1.25×	+1.24×
	HE	40373 ms	+1.91×	−1.01×	+1.01×
	AE	31100 ms	+2.64×	−1.59×	+1.32×
	FA	25539 ms	+2.94×	–	+1.61×

Table 6.1: Overview of experiments on the end-to-end query execution times on the Join Order Benchmark, and the chain and cycle benchmark, comparing the robust plans according to FS, SS, and FI, against the heuristic and adaptive plans HE and AE, the estimated optimal plans EO, and the fastest plans FA.

starts with the estimated optimal plan. In the estimated optimal plan, the first operators can already cause severe cardinality estimation errors, which cannot be compensated by future plan switches. The robustness metrics can detect such situations and assign a bad robustness value to the corresponding plans. We discuss in Section 7.1.3 that a combination of adaptive execution and robust plan selection could be promising. The robustness values could be considered in the initial plan search and all re-optimizations.

		Σ time	best speedup	worst regression	average
Star	EO	178520 ms	–	–	–
	FS	182693 ms	+1.14×	−1.49×	−1.02×
	SS	177251 ms	+1.23×	−1.48×	+1.01×
	FI	188334 ms	+1.21×	−1.48×	−1.05×
	HE	177231 ms	+1.16×	−1.08×	+1.01×
	AE	174444 ms	+2.25×	−1.18×	+1.02×
	FA	161987 ms	+1.47×	–	+1.10×
Snowflake	EO	53793 ms	–	–	–
	FS	54437 ms	+1.53×	−2.07×	−1.01×
	SS	51579 ms	+1.91×	−1.36×	+1.04×
	FI	53327 ms	+1.78×	−1.38×	+1.01×
	HE	49442 ms	+3.39×	−1.01×	+1.08×
	AE	43307 ms	+3.04×	−1.11×	+1.24×
	FA	44843 ms	+2.11×	–	+1.20×
Random	EO	82001 ms	–	–	–
	FS	75644 ms	+2.14×	−1.60×	+1.08×
	SS	74951 ms	+2.27×	−1.22×	+1.09×
	FI	78922 ms	+2.04×	−1.33×	+1.04×
	HE	63393 ms	+2.67×	−1.01×	+1.29×
	AE	43739 ms	+5.19×	−1.39×	+1.87×
	FA	56273 ms	+4.07×	–	+1.46×

Table 6.2: Overview of experiments on the end-to-end query execution times on the star, snowflake and random benchmark, comparing the robust plans according to FS, SS, and FI, against the heuristic and adaptive plans HE and AE, the estimated optimal plans EO, and the fastest plans FA.

From this comparison we conclude that adaptive execution is superior to robust plan selection. The adaptive execution achieves more and stronger speedups, as well as less severe degradations compared to the robust plan selection. One root cause for the dominance of adaptive execution is the utilization of true cardinalities, while the robust plan selection is solely based on estimated cardinalities. Furthermore, the adaptive execution tries to steadily improve the true plan cost, and hence also the query execution time of the initially es-

timated cheapest plan. In contrast, the robust plan selection tries to identify a query execution plan, whose cost is less sensitive to cardinality estimation errors. Consequently, the robust plan can be slower at good cardinality estimates than the estimated optimal plan. Nevertheless, our experiments illustrate that robust plan selection can improve plan robustness and end-to-end query execution time in case of cardinality estimation errors. In case only the optimizer can be modified, we recommend our robust plan selection strategy using one of the three robustness metrics of Chapter 4. If high engineering costs are acceptable, we recommend the architecture and execution strategy that we showcase in our adaptive query processor in Chapter 5.

7 FUTURE DIRECTIONS AND CONCLUSION

The contributions we make in this work address existing and future challenges of query processing. Our approaches for the detection and compensation of sub-optimal query execution plans, caused by cardinality estimation errors, creates a new momentum for robust and adaptive query processing. In this chapter, we discuss the future directions and the potential impact of our work.

7.1 FUTURE DIRECTIONS

Our formal and conceptual studies as well as the detailed experimental evaluations build a solid foundation for the future work on optimality ranges, robustness of query execution plans, and adaptive query execution. For each of our contributions we see some promising future directions, which we discuss next.

7.1.1 *Optimality Ranges*

To calculate an optimality range for one variable cardinality, it is theoretically sufficient to consider only two alternative query execution plans, i.e., the ones that intersect with the optimal query execution plan at the lower bound and at the upper bound. To identify those two plans, we have to enumerate a considerably larger amount of query execution plans than just two. Enumerating a smaller number of plans would increase the applicability of optimality ranges, e.g., as re-optimization criterion for ad-hoc queries. From our point of view, the most promising approach is to create further pruning strategies.

Some applications do not need precise optimality ranges. Another but weaker achievement would be the calculation of almost precise optimality ranges. Ideally, a considerably smaller number of query execution plans has to be enumerated to derive the almost precise optimality ranges. In an ideal solution, there should be furthermore a configuration parameter for the precision of the almost precise optimality ranges. Further variants of optimality ranges could state that a plan is optimal in a smaller range than the actual optimality range, so that it is guaranteed that there is no better plan in the smaller range. Compared to precise optimality ranges, the plan could still be optimal beyond the smaller range. The opposite of smaller ranges is also promising. Larger ranges than the actual optimality range can state that there is for sure a better plan beyond the larger range, although there might be even better plans within the larger range.

In Chapter 3 we describe optimality range calculation as post optimization step. Another promising idea is to derive optimality ranges during the initial plan search. Furthermore, future work can extended optimality range calculation for multi-dimensional Parametric Cost Functions, and non-linear Parametric Cost Functions.

7.1.2 *Robustness Metrics and Robust Plan Selection*

There is another approach for robust plan selection, which could directly utilize our calculation algorithm for optimality ranges. During the initial plan enumeration, optimality ranges could be derived for each sub-plan. To identify a robust plan, the plan table of the query optimizer does not necessarily contain the estimated cheapest plan for each plan class, but the plan with the largest optimality range. Combining the sub-plans with the largest optimality ranges may result in an overall more robust plan. Ideally, such a robust plan can give strong guarantees for robustness.

To keep the plan robustness analysis cheap, we consider only one intermediate result cardinality at a time, i.e., create one-dimensional Parametric Cost Functions. In the next step, we combine the information we derive from all single intermediate result cardinalities. The plan robustness analysis is actually a multi-dimensional problem. To consider the impact of different cardinality estimation errors on each other, it would be furthermore interesting to create multi-dimensional Parametric Cost Functions, and analyze their slope behavior.

It is furthermore promising to add additional features to our robustness metrics, to consider further risks of rapid cost changes. A different selection algorithm for the robust plan candidates may also result in even more robust query execution plans. Our approach for robustness metrics can be also extended for cost function parameters other than cardinality, to consider the robustness with respect to, e.g., resource consumption or distribution. Furthermore, an improved support for arbitrary shapes of Parametric Cost Functions in the cardinality-slope and selectivity-slope metric is useful.

7.1.3 *Mid-Query Re-Optimization*

In our adaptive query processor, we collect only the true cardinalities of intermediate results. To further improve the end-to-end query execution time of adaptive plans, it may amortize to collect more runtime statistics, such as true distinct count, true maximum, or true minimum of some interesting columns. The additional effort could be compensated through further improvements of the plan quality. It may be sufficient to just estimate those statistics at runtime, e.g., through sketches for distinct count [64], because the true value might not be necessary. Further statistics can also help to reduce the num-

ber of true cost degradations we observe in Section 5.5.1, which are caused by cardinality estimation errors during the re-optimizations.

Although our experiments in Section 5.5.4 reveal that the overall effort for plan adaptions is similar to the effort for the initial plan search, it is an improvement to further reduce the adaption effort. The experiments in Section 5.5.3 reveal that there are re-optimizations, which do not find a better plan. Avoiding those re-optimization, maybe through cheaper optimality ranges, would be an improvement.

The most performance degradations we observe in our experiments in Section 5.5.2 are a consequence of cost model inaccuracies. In our experiments, the root causes for cost model inaccuracies are mostly skewed input columns of the hash join build sides. A skewed input column can increase the cost of a hash join build side, compared to a similar input column with uniform value distribution. Since we use the C_{mm} [93] cost function, which does not cover skew on the input column of the hash join build side, there are adaptive plans that should perform better than the estimated optimal plan according to the true cost improvement, but perform worse with respect to query execution time. One could argue that a better cost function, which considers additional parameters could prevent such effects. We argue that also the operators have to become more predictable for the query optimizer. Ideally, the performance of operators should only depend on some basic parameters such as input cardinality.

We explain in Section 5.4.1 that our adaptive query processor executes only one pipeline of a query execution plan at a time. Theoretically multiple pipelines could be executed at a time. Executing only one pipeline at a time creates space for pipeline selection strategies. Corresponding to the work of Neumann and Galindo-Legaria [69], it can be promising to choose a pipeline, whose breaker contains an intermediate result with a strong impact on the plan choice.

Additional future direction may combine adaptive execution with other query processing approaches. Our experimental comparison of robust plan selection and adaptive execution in Section 6.2 reveals that robust plans can avoid early estimation errors, which cannot be compensated through plan switches. Instead of starting with the estimated cheapest query execution plan, our adaptive query processor could also start with a more robust query execution plan. During the re-optimizations, the adaptive query processor could consider the robustness values of the plan alternatives again. Adaptive execution could be also combined with plan caching. Instead of caching the entire plan, just the instantiated or compiled pipelines can be cached. Our adaptive execution strategy could be also used for large join queries [99]. A large join query could be re-optimized after a few operators to improve the not-yet-executed query parts. Since only the not-yet-executed query parts are considered, the optimization effort decreases with each re-optimization.

7.2 CONCLUSION

In this work, we make contributions in the field of Robust and Adaptive Query Processing. All our contributions are related to the detection and compensation of sub-optimal query execution plans, caused by cardinality estimation errors. We argue in Chapter 1 that unsolved problems and limitations of existing work to detect and compensate sub-optimal plans should rather be addressed, instead of trying to further improve cardinality estimation. We demonstrate that existing heuristics cannot characterize the intermediate result cardinalities, for which query execution plans are optimal, and present an algorithm to calculate precise optimality ranges. Further valuable and generic building blocks for Robust and Adaptive Query Processing are our metrics to quantify the plan robustness. While the optimality ranges improve the detection of sub-optimal query execution plans, our robustness metrics quantify the robustness of query execution plans with respect to cardinality estimation errors. The corresponding robust plan selection strategy can improve the end-to-end query execution time of queries with cardinality estimation errors by up to 3.49×.

Our adaptive query processor combines the lessons learned from existing work on Mid-Query Re-Optimization, and showcases that runtime feedback considerably improves the query performance, introducing only a negligible risk of performance degradations. Our experiments reveal that true cardinalities are superior to estimated cardinalities, and that the overall effort for plan adaptions is similar to an initial optimization. In the end, our adaptive query processor can improve the end-to-end query execution times by up to 5.19×.

Our work confirms the observation of Lohman [94] that "*robust and adaptable query plans are superior to optimal ones*". Like Leis et al. [78, 93] demonstrate that simple cost functions are sufficient for query optimization, we demonstrate that basic statistics are sufficient for cardinality estimation, when cardinality estimation errors are compensated with true cardinalities collected at query execution time.

Since our work improves the query performance, it has an impact on existing database systems, which have to process an ever-growing amount of data, as well as queries with increasing complexity. The impact of our work becomes more prominent, the more database systems support Hybrid Transactional/Analytical Processing. While the availability and freshness of statistics is limited in Hybrid Transactional/Analytical Processing, the requirements on query execution plan quality and performance are similar to conventional database systems. In this work, we presents new promising approaches to improve the processing of complex queries in setups where only basic statistics exist. Our contributions are not limited to Hybrid Transactional/Analytical Processing, and can also speedup queries in conventional analytical database systems.

BIBLIOGRAPHY

[1] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Communications of the ACM* 13.6 (June 1970).

[2] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '74. ACM, 1974.

[3] Raymond A. Lorie. "XRM - An Extended (N-ary) Relational Memory." In: *IBM Research Report* G320-2096 (1974).

[4] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access Path Selection in a Relational Database Management System." In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD '79. Boston, Massachusetts, USA: ACM, 1979.

[5] Philip A. Bernstein and Nathan Goodman. "Multiversion Concurrency Control – Theory and Algorithms." In: *ACM Transactions on Database Systems* 8.4 (Dec. 1983).

[6] Theo Haerder and Andreas Reuter. "Principles of Transaction-oriented Database Recovery." In: *ACM Computing Surveys* 15.4 (Dec. 1983).

[7] George P. Copeland and Setrag N. Khoshafian. "A Decomposition Storage Model." In: *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*. SIGMOD '85. ACM, 1985.

[8] R. W Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, Inc., 1986.

[9] Lothar F. Mackert and Guy M. Lohman. "R* Optimizer Validation and Performance Evaluation for Distributed Queries." In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB '86. Morgan Kaufmann Publishers Inc., 1986.

[10] Lothar F. Mackert and Guy M. Lohman. "R* Optimizer Validation and Performance Evaluation for Local Queries." In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. SIGMOD '86. ACM, 1986.

[11] Jim Gray and Franco Putzolu. "The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time." In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD '87. ACM, 1987.

[12] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. Addison-Wesley Longman Publishing Co., Inc., 1989.

[13] Y. E. Ioannidis and Younkyung Kang. "Randomized Algorithms for Optimizing Large Join Queries." In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD '90. ACM, 1990.

[14] Kiyoshi Ono and Guy M. Lohman. "Measuring the Complexity of Join Enumeration in Query Optimization." In: *Proceedings of the 16th International Conference on Very Large Data Bases*. VLDB '90. Morgan Kaufmann Publishers Inc., 1990.

[15] Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. "A Genetic Algorithm for Database Query Optimization." In: *In Proceedings of the fourth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, 1991.

[16] Yannis E. Ioannidis and Stavros Christodoulakis. "On the Propagation of Errors in the Size of Join Results." In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. SIGMOD '91. ACM, 1991.

[17] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. "Query Optimization in the IBM DB2 Family." In: *IEEE Data Engineering Bulletin* 16 (1993).

[18] Goetz Graefe. "Query Evaluation Techniques for Large Databases." In: *ACM Computing Surveys* 25.2 (June 1993).

[19] Goetz Graefe and William J. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search." In: *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, 1993.

[20] Allen Van Gelder. "Multiple Join Size Estimation by Virtual Domains (Extended Abstract)." In: *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '93. ACM, 1993.

[21] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. "Fast, Randomized Join-Order Selection - Why Use Transformations?" In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. Morgan Kaufmann Publishers Inc., 1994.

[22] Goetz Graefe. "Volcano - An Extensible and Parallel Query Evaluation System." In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (Feb. 1994).

[23] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. "Query Optimization by Predicate Move-Around." In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. Morgan Kaufmann Publishers Inc., 1994.

[24] Arun Swami and K. Bernhard Schiefer. "On the Estimation of Join Result Sizes." In: *Proceedings of the 4th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '94. Springer-Verlag New York, Inc., 1994.

[25] Weipeng P. Yan and Per-Ake Larson. "Performing Group-By Before Join." In: *Proceedings of the Tenth International Conference on Data Engineering*. IEEE Computer Society, 1994.

[26] Sophie Cluet and Guido Moerkotte. "On the Complexity of Generating Optimal Left-Deep Processing Trees with Cross Products." In: *Proceedings of the 5th International Conference on Database Theory*. ICDT '95. Springer-Verlag, 1995.

[27] Bennet Vance and David Maier. "Rapid Bushy Join-order Optimization with Cartesian Products." In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. ACM, 1996.

[28] Jim Gray and Goetz Graefe. "The Five-minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb." In: *SIGMOD Record* 26.4 (Dec. 1997).

[29] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. "The Complexity of Transformation-Based Join Enumeration." In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. VLDB '97. Morgan Kaufmann Publishers Inc., 1997.

[30] Leonidas Fegaras. "A New Heuristic for Optimizing Large Queries." In: *Proceedings of the 9th International Conference on Database and Expert Systems Applications*. DEXA '98. Springer-Verlag, 1998.

[31] Navin Kabra and David J. DeWitt. "Efficient Mid-query Reoptimization of Sub-optimal Query Execution Plans." In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. SIGMOD '98. Seattle, Washington, USA: ACM, 1998.

[32] 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. ACM, 2000.

[33] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. "Towards Estimation Error Guarantees for Distinct Values." In: *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '00. ACM, 2000.

[34] Donald Kossmann. "The State of the Art in Distributed Query Processing." In: *ACM Computing Surveys* 32.4 (Dec. 2000).

[35] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. "Weaving Relations for Cache Performance." In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. Morgan Kaufmann Publishers Inc., 2001.

[36] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. "LEO - DB2's LEarning Optimizer." In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. Morgan Kaufmann Publishers Inc., 2001.

[37] Arvind Hulgeri and S. Sudarshan. "Parametric Query Optimization for Linear and Piecewise Linear Cost Functions." In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. VLDB Endowment, 2002.

[38] Arvind Hulgeri and S. Sudarshan. "AniPQO: Almost Non-intrusive Parametric Query Optimization for Nonlinear Cost Functions." In: *Proceedings of the 29th International Conference on Very Large Data Bases*. VLDB '03. VLDB Endowment, 2003.

[39] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. "Robust Query Processing Through Progressive Optimization." In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France, 2004.

[40] Brian Babcock and Surajit Chaudhuri. "Towards a Robust Query Optimizer: A Principled and Practical Approach." In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. ACM, 2005.

[41] Shivnath Babu, Pedro Bizarro, and David DeWitt. "Proactive Re-optimization." In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland, USA: ACM, 2005.

[42] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *Second Biennial Conference on Innovative Data Systems Research*. CIDR '05. 2005.

[43] Thomas Neumann, Sven Helmer, and Guido Moerkotte. "On the Optimal Ordering of Maps and Selections Under Factorization." In: *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05. IEEE Computer Society, 2005.

[44] Guido Moerkotte and Thomas Neumann. "Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees Without Cross Products." In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. VLDB Endowment, 2006.

[45] 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. VLDB Endowment, 2007.

[46] David DeHaan and Frank Wm. Tompa. "Optimal Top-down Join Enumeration." In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. ACM, 2007.

[47] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. "Adaptive Query Processing." In: *Foundations and Trends in Databases* 1.1 (Jan. 2007).

[48] Goetz Graefe. "The Five-minute Rule Twenty Years Later, and How Flash Memory Changes the Rules." In: *Proceedings of the 3rd International Workshop on Data Management on New Hardware*. DaMoN '07. ACM, 2007.

[49] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. "Adaptively Reordering Joins during Query Execution." In: *2007 IEEE 23rd International Conference on Data Engineering*. ICDE'07. IEEE Computer Society, 2007.

[50] M. Tamer Ozsu. *Principles of Distributed Database Systems*. 3rd. Prentice Hall Press, 2007.

[51] Renzo Angles and Claudio Gutierrez. "Survey of Graph Database Models." In: *ACM Computing Surveys* 40.1 (Feb. 2008).

[52] Tyson Condie, David Chu, Joseph M. Hellerstein, and Petros Maniatis. "Evita Raced: Metacompilation for Declarative Networks." In: *Proceedings of the VLDB Endowment* 1.1 (Aug. 2008).

[53] Harish D., Pooja N. Darera, and Jayant R. Haritsa. "Identifying Robust Plans Through Plan Diagram Reduction." In: *Proceedings of the VLDB Endowment* 1.1 (Aug. 2008).

[54] Guido Moerkotte and Thomas Neumann. "Dynamic Programming Strikes Back." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. ACM, 2008.

[55] Pedro Bizarro, Nicolas Bruno, and David J. DeWitt. "Progressive Parametric Query Optimization." In: *IEEE Transactions on Knowledge and Data Engineering* 21.4 (Apr. 2009).

[56] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. "Exact Cardinality Query Optimization for Optimizer Testing." In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009).

[57] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[58] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. "Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors." In: *Proceedings of the VLDB Endowment* 2.1 (Aug. 2009).

[59] M. Abhirama, Sourjya Bhaumik, Atreyee Dey, Harsh Shrimal, and Jayant R. Haritsa. "On the Stability of Plan Costs and the Costs of Plan Stability." In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010).

[60] Goetz Graefe, Arnd Christian König, Harumi A. Kuno, Volker Markl, and Kai-Uwe Sattler, eds. *Robust Query Processing (Dagstuhl Seminar 10381)*. Dagstuhl Seminar Proceedings. Leibniz-Zentrum für Informatik, Germany: Schloss Dagstuhl, 2010.

[61] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. "HYRISE: A Main Memory Hybrid Storage Engine." In: *Proceedings of the VLDB Endowment* 4.2 (Nov. 2010).

[62] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots." In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. IEEE Computer Society, 2011.

[63] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems." In: *Proceedings of the VLDB Endowment* 5.10 (June 2012).

[64] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches." In: *Foundations and Trends in Databases* 4.1–3 (Jan. 2012).

[65] Goetz Graefe, Wey Guy, Harumi A. Kuno, and Glenn N. Paulley. "Robust Query Processing (Dagstuhl Seminar 12321)." In: *Dagstuhl Reports* 2.8 (2012).

[66] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. "Multi-core, Main-memory Joins: Sort vs. Hash Revisited." In: *Proceedings of the VLDB Endowment* 7.1 (Sept. 2013).

[67] Pit Fender and Guido Moerkotte. "Counter Strike: Generic Top-down Join Enumeration for Hypergraphs." In: *Proceedings of the VLDB Endowment* 6.14 (Sept. 2013).

[68] Guido Moerkotte, Pit Fender, and Marius Eich. "On the Correct and Complete Enumeration of the Core Search Space." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. ACM, 2013.

[69] Thomas Neumann and César A. Galindo-Legaria. "Taking the Edge off Cardinality Estimation Errors using Incremental Execution." In: *Datenbanksysteme für Business, Technologie und Web*. BTW '13. 2013.

[70] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. "Memory-efficient Hash Joins." In: *Proc. VLDB Endow.* 8.4 (Dec. 2014).

[71] Anshuman Dutt and Jayant R. Haritsa. "Plan Bouquets: Query Processing Without Selectivity Estimation." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. ACM, 2014.

[72] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven Parallelism: A NUMA–aware Query Evaluation Framework for the Many-core Age." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. ACM, 2014.

[73] Guy M Lohman. "Is query optimization a "solved" problem." In: *Proceedings of the Workshop on Database Query Optimization*. Oregon Graduate Center Comp. Sci. Tech. Rep. 2014.

[74] Guido Moerkotte. *Building Query Compilers*. unpublished. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`, 2014.

[75] Thomas Neumann. "Engineering High-performance Database Engines." In: *Proc. VLDB Endow.* 7.13 (Aug. 2014).

[76] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. "Scaling up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling." In: *6th TPC Technology Conference on Performance Evaluation and Benchmarking*. TPCTC '14. Hangzhou, China: Springer International Publishing, 2014.

[77] Khaled H Alyoubi, Sven Helmer, and Peter T Wood. "Ordering selection operators using the minmax regret rule." In: *arXiv preprint arXiv:1507.08257* (2015).

[78] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proceedings of the VLDB Endowment* 9.3 (Nov. 2015).

[79] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. "Cache-Efficient Aggregation: Hashing Is Sorting." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015.

[80] Shaoyi Yin, Abdelkader Hameurlain, and Franck Morvan. "Robust Query Optimization Methods With Respect to Estimation Errors: A Survey." In: *SIGMOD Record* 44.3 (Dec. 2015).

[81] Khaled Hamed Alyoubi. "Database query optimisation based on measures of regret." PhD thesis. Birkbeck, University of London, 2016.

[82] Fabian Hüske. "Specification and optimization of analytical data flows." PhD thesis. TU Berlin, 2016.

[83] Fisnik Kastrati and Guido Moerkotte. "Optimization of Conjunctive Predicates for Main Memory Column Stores." In: *Proceedings of the VLDB Endowment* 9.12 (Aug. 2016).

[84] Mengmeng Liu, Zachary G. Ives, and Boon Thau Loo. "Enabling Incremental Query Re-Optimization." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. ACM, 2016.

[85] Stefan Schuh, Xiao Chen, and Jens Dittrich. "An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. ACM, 2016.

[86] Reza Sherkat et al. "Page As You Go: Piecewise Columnar Access In SAP HANA." In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. ACM, 2016.

[87] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. "The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy." In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures 2017*. ADMS '17. 2017.

[88] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. "Distributed Join Algorithms on Thousands of Cores." In: *Proceedings of the VLDB Endowment* 10.5 (Jan. 2017).

[89] Transaction Processing Performance Council. *TPC Benchmark H (Decision Support) Standard Specification Revision 2.17.2*. `http://www.tpc.org`. 2017.

[90] Marius Eich, Pit Fender, and Guido Moerkotte. "Efficient generation of query plans containing group-by, join, and groupjoin." In: *The VLDB Journal* (Aug. 2017).

[91] Goetz Graefe, Renata Borovica-Gajic, and Allison Lee. "Robust Performance in Database Query Processing (Dagstuhl Seminar 17222)." In: *Dagstuhl Reports* 7.5 (2017).

[92] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. Morgan Kaufmann Publishers Inc., 2017.

[93] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Query optimization through the looking glass, and what we found running the Join Order Benchmark." In: *The VLDB Journal* (Sept. 2017).

[94] Guy Lohman. "Query Optimization–Are We There Yet?" In: *Datenbanksysteme für Business, Technologie und Web*. BTW '17. Gesellschaft für Informatik, Bonn, 2017.

[95] Norman May, Alexander Böhm, and Wolfgang Lehner. "SAP HANA – The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads." In: *Datenbanksysteme für Business, Technologie und Web*. BTW '17. Gesellschaft für Informatik, Bonn, 2017.

[96] Renzo Angles et al. "G-CORE: A Core for Future Graph Query Languages." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. ACM, 2018.

[97] Michael Brendle. "A Robustness Metric for Relational Query Execution Plans." MA thesis. University of Konstanz, 2018.

[98] Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines." In: *Proceedings of the 14th International Workshop on Data Management on New Hardware*. DAMON '18. ACM, 2018.

[99] Thomas Neumann and Bernhard Radke. "Adaptive Optimization of Very Large Join Queries." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. ACM, 2018.

[100] Florian Wolf, Norman May, Paul R. Willems, and Kai-Uwe Sattler. "On the Calculation of Optimality Ranges for Relational Query Execution Plans." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018.

[101] Florian Wolf, Michael Brendle, Norman May, Paul R. Willems, Kai-Uwe Sattler, and Michael Grossniklaus. "Robustness Metrics for Relational Query Execution Plans." In: *Proceedings of the VLDB Endowment 11.11*. VLDB '18. Rio de Janeiro, Brazil: VLDB Endowment, 2018.

ERKLÄRUNG

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Bei der Auswahl und Auswertung folgenden Materials haben mir die nachstehend aufgeführten Personen in der jeweils beschriebenen Weise unentgeltlich geholfen:

1. Dr. Norman May, Ko-Autor von [100, 101], Ansprechpartner bei SAP
2. Dr. Paul R. Willems, Ko-Autor von [100, 101], Ansprechpartner bei SAP
3. Michael Brendle, Ko-Autor von [101], Studentische Hilfskraft
4. Prakriti Bhardwaj, Studentische Hilfskraft

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß § 7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

Heidelberg, September 2018

Florian Wolf