Informatics

Integration von Strukturbasierter Abfrageoptimierung in NewSQL-Datenbanken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Michael Martinek, BSc
Matrikelnummer 11809624

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Univ.Ass. Dipl.-Ing. Alexander Selzer, BSc

Wien, 2. Dezember 2023

Michael Martinek   Reinhard Pichler

Informatics

Integration of Structure Guided Query Optimization into NewSQL Databases

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Michael Martinek, BSc
Registration Number 11809624

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Univ.Ass. Dipl.-Ing. Alexander Selzer, BSc

Vienna, 2nd December, 2023

Michael Martinek   Reinhard Pichler

Erklärung zur Verfassung der Arbeit

Michael Martinek, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Dezember 2023

Michael Martinek

Danksagung

Ich danke meinem Betreuer Professor Reinhard Pichler für seine Unterstützung während der Diplomarbeit, sowie die investierte Zeit. Besonders in der Themenwahl hat er mich in die richtige Richtung geleitet, und mit guten Materialien und Quellen für einen Start in das Thema unterstützt. Außerdem war sein Feedback, zu jedem Zeitpunkt der Arbeit, immer konstruktiv und hilfreich.

Zudem danke ich Alexander Selzer, der diese Diplomarbeit mitbetreut hat. Er hat mich im praktischen Einstieg in das Thema, sowie in der Orientierung bezüglich der Implementierung unterstützt. Außerdem hat er mir in der Auswahl der geeigneten Infrastruktur für die Evaluierungen geholfen, und mich mit dem TU.it dataLAB Team der TU Wien in Kontakt gebracht.

Vielen Dank an das TU.it dataLAB Team der TU Wien für die Bereitstellung von Rechenleistung und Infrastruktur, welche auf meine Anforderungen angepasst wurde, um die Evaluierungen durchzuführen. Insbesondere möchte ich meinen Kontaktpersonen Natalie Kamenik, Dieter Kvasnicka und Giovanna Roda danken.

Abschließend möchte ich allen danken, die mich während meiner Diplomarbeit und des ganzen Studiums unterstützt haben.

Acknowledgements

I would like to thank my advisor Professor Reinhard Pichler for his support and guidance during the creation of this thesis and the time invested. Especially in the process of choosing a topic, he was very helpful and pushed me in the right direction, and provided me with great material and sources to get started with the topic. Furthermore, his feedback during the whole time, from choosing a topic to finishing the thesis was always constructive and helpful.

Moreover, I would like to thank Alexander Selzer, who was my co-advisor during this thesis. He supported me getting started into the topic from a practical view, and provided me some guidance for the implementation. Also, he helped me finding a suitable infrastructure to conduct evaluations of the implementation, and connected me with the TU.it dataLAB team of the TU Wien.

Thanks to the TU.it dataLAB Team of the TU Wien for providing computing capacity and infrastructure, tailored towards my needs, to conduct evaluations. In particular, I would like to thank my contacts at the team, Natalie Kamenik, Dieter Kvasnicka, and Giovanna Roda.

Finally, I would like to thank everyone who supported me during my thesis, but also throughout my whole study.

Kurzfassung

Seit jeher sind wir bestrebt die Leistung, Effizienz und Funktionalität von Datenbanksystemen zu verbessern. Dies beinhaltet auch Techniken zur Bewältigung des andauernden Anstieges der Datenmengen in Datenbanken. Ein Schritt zur Bewältigung dieser großen Datenmengen war die Einführung von NoSQL, und folgend für das relationale Datenmodell, sowie die Anforderungen von relationalen Datenbanken, NewSQL Datenbanken.

Gleichzeitig müssen die Methoden zur Datenabfrage an die ständig steigenden Datenmengen angepasst werden. Ein derzeit kaum genutzter Ansatz ist strukturbasierte Abfrageoptimierung, welche die Struktur einer Abfrage nutzt, um einen optimalen Join-Baum, basierend auf einer Zerlegung der Abfrage, zu bilden. Darüber hinaus eliminiert der in der Datenbanktheorie bekannte Algorithmus von Yannakakis redundante Tupel, die nicht zum Endergebnis beitragen, durch zwei Runden Semi-Joins. Eine kürzlich durchgeführte Analyse hat gezeigt, dass die meisten realistischen Datenbankabfragen azyklisch oder fast azyklisch sind, so dass sie mit dem Algorithmus von Yannakakis gut verarbeitet werden können. Diese Technik löst zwei der größten Probleme bei der Auswertung von Datenbankabfragen: die Suche nach einer optimalen Join-Reihenfolge und die Vermeidung einer Explosion von Zwischenergebnissen bei der Auswertung einer konjunktiven Abfrage. Trotz der bewiesenen Leistungsvorteile für einige Abfragen wird dieser Algorithmus in der Praxis jedoch von keinem der großen Produktionsdatenbanksysteme verwendet.

Das Ziel dieser Arbeit ist es, diese Lücke zwischen theoretisch möglicher Leistung und praktisch genutzten Ansätzen zu schließen, indem wir strukturbasierte Abfrageoptimierung in das NewSQL DBMS TiDB integrieren. TiDB zählt zu den bekanntesten Open-Source NewSQL DBMS. Die empirischen Auswertungen zeigten Leistungsvorteile für einige Abfragen durch strukturbasierte Abfrageauswertung. 40% unserer Full-Enumeration Testabfragen, sowie 30% unserer 0MA Testabfragen hatten mit unserer Implementierung eine geringere Laufzeit als mit herkömmlicher Abfrageauswertung. Der Bedarf an Hauptspeicher ist durch die temporäre Speicherung von Zwischenergebnissen während der Abfrageausführung leicht angestiegen. Besonderes Augenmerk fällt auf die Klasse der Full-Enumeration Abfragen, deren Ergebnis nicht leer ist. Unsere Implementierung hat die Mehrzahl dieser Instanzen mit niedrigerer Laufzeit als die konventionelle Abfrageauswertung gelöst. Die Ergebnisse verdeutlichen die Nützlichkeit der Integration von strukturbasierter Abfrageoptimierung als zusätzliche Abfrageoptimierungsoption in einem DBMS.

Abstract

Ever since database systems exist, we strive to improve performance, efficiency, and functionalities of databases, including techniques to handle the ongoing trend of rising amounts of data in databases. One step towards dealing with these big amounts of data was the introduction of NoSQL, and subsequently for the relational data model and requirements of RDBMS, NewSQL databases.

In parallel, querying of data needs to be adapted to handle the rising amounts of data. One possible approach, which is hardly used at all, is structure guided query optimization, leveraging the structure of a query to build an optimal join tree based on a decomposition of the query, without the need to find an optimal join ordering through traditional query optimization. Additionally, Yannakakis' algorithm, which is well known in database theory, eliminates redundant tuples not contributing to the end result by conducting two rounds of semi-joins, before the actual join of the remaining tuples. A recent analysis showed that most real-world database queries are acyclic, or at least almost acyclic, which can be handled well by Yannakakis' algorithm. This technique solves two of the most urgent problems in database query evaluation, finding an optimal join order, and avoiding an explosion of intermediate results during evaluation of a conjunctive query. However, no major production database system uses this algorithm in practice, despite these proven performance benefits for some queries.

The aim of this thesis is to fill this gap between theoretically possible performance and practically used approaches, and integrate structure guided query optimization into the NewSQL DBMS TiDB, which lies among the most popular Open-Source NewSQL DBMS. Our empirical evaluations showed performance benefits for some queries through structure guided query evaluation. 40% of our full enumeration test queries, and 30% of our 0MA test queries were faster with our implementation. The memory usage slightly increased, due to the need for additional temporary tables to store intermediate results. Particular attention falls into the class of full enumeration queries whose result is not empty, for which the majority of instances were faster with our implementation. The results clearly show the benefit of integrating structure guided query optimization, as additional query optimization option, deeply and directly into a DBMS.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Problem Statement and Motivation 1
1.2 Aim of the Thesis 2
1.3 Methodology 3
1.4 State of the Art 4
1.5 Results 5
1.6 Structure of the Thesis 6

2 Emergence of NewSQL **7**
2.1 Relational Data Model - RDBMS - SQL 7
2.2 Path to NewSQL 11
2.3 TiDB 15

3 Structure Guided Query Optimization **25**
3.1 Queries 26
3.2 Query Decomposition 30
3.3 Yannakakis' Algorithm - Structure Guided Execution 35
3.4 Special Queries: 0MA 40

4 Integrating Structure Guided Query Optimization into TiDB **41**
4.1 Current Database Implementation 42
4.2 Integrating Structure Guided Query Optimization 47
4.3 Final Architecture 61

5 Experimental Evaluation **63**
5.1 Methodology 63
5.2 Feature Evaluations 66
5.3 Main Evaluation 74

5.4 Discussion 79

6 Conclusion **83**

6.1 Future Work 84

List of Figures **85**

List of Tables **87**

List of Code Listings **89**

Acronyms **91**

Bibliography **93**

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

Ever since databases exist, we strive to improve their performance and abilities. Whether it is the amount of data which can be worked on, the resilience of the database against failures and scalability, or the querying performance. All of those aspects were improved in the past, and are going to be improved in the future.

As the amount of data used in databases rose, new database architectures, different from the traditional relational Database Management System (DBMS), which are often bound to one hardware system, had to be developed. First, Not Only SQL (NoSQL) emerged and solved many problems regarding scalability and failure resistance. However, a lot of NoSQL databases did not provide the necessary consistency guarantees. This was one reason why NewSQL databases came up. NewSQL uses a relational data model, together with a scalable and failure resistant storage backend, running in a cluster architecture. Furthermore, Structured Query Language (SQL) could be used as a familiar query language in this architecture. However, with more data in the database, queries and results are potentially bigger. This leads to the problem of efficient evaluation of these big queries, especially when they include joins of many relations.

Arguably, the most fundamental form of queries are conjunctive queries, defined as a *select ... from ... where ...* query, in which the where clause only contains conjunctions of equalities. The query planners of most modern relational DBMS, including NewSQL databases, use statistical information on the data to determine the best join order to join pairs of relations or intermediate results together. In this approach, the intermediate results possibly contain tuples not contributing to the end result of the query (referred to as dangling tuples). This can lead to an exponential blowup of intermediate results. Furthermore, as finding an optimal join order is NP-complete, often times, only approximate, suboptimal solutions for the ordering are determined, which additionally contributes to this problem [NR18].

For acyclic and almost acyclic conjunctive queries, the explosion of intermediate results could be prevented by using a join approach, which is well known in the field of database theory, Yannakakis' algorithm [Yan81], also referred to as structure guided query evaluation [GLL+23]. In the process of joining, two additional passes are introduced, which eliminate tuples not contributing to the end result, i.e. preventing an explosion of intermediate results. Additionally, this approach naturally yields a good join order. With this algorithm, the decision if there are result tuples for a query can be made in polynomial time, and enumerating the output can be done in output polynomial time. This is especially interesting as a recent paper by Fischl et al. [FGLP21] analyzed hundreds of real-world and benchmark database queries for their acyclicity. The result of the analysis found that most real-world database queries are acyclic, or at least almost acyclic, which can be handled well by Yannakakis' algorithm. However, no major production database system uses this algorithm in practice, despite these proven performance benefits for some queries. This divergence between the theoretically possible performance and the currently used practical approaches has to be overcome.

1.2 Aim of the Thesis

The goal of this thesis is to fill this gap and find out whether Yannakakis' theoretically advantageous algorithm plays to its strengths in practice. To this end, we will integrate the execution of acyclic and almost acyclic conjunctive queries according to Yannakakis' algorithm deep into the NewSQL DBMS TiDB (`https://github.com/pingcap/tidb`).

Modern DBMS already optimize queries quite well, especially as the optimizers can use parallelized computation to simultaneously handle multiple possible optimization paths. However, even when an optimal join order is found, there can still be an explosion of intermediate results, due to dangling tuples. Therefore, we integrate Yannakakis' algorithm deep into a NewSQL DBMS, namely TiDB. From a theoretical standpoint, by eliminating redundant tuples before the actual join, the join should be sped up. Following, we expect to see a performance optimization for these queries. However, it should be noted that Yannakakis' algorithm also introduces additional costs caused by the addition of two rounds of semi-joins and materializing their results during query computation. It therefore remains to be seen if there are actual performance improvements. Additionally, it will be interesting to see if there are certain types of queries, for example the computation of extreme values, as shown by Gottlob et al. in [GLL+23], which yield better results than others, compared to normal execution.

This thesis is the first to directly and deeply integrate Yannakakis' algorithm into a NewSQL DBMS, creating an extended DBMS that makes use of a proven theoretical approach, which should solve two of the most urgent problems in database query evaluation, finding an optimal join order, and avoiding an explosion of intermediate results. Extensive experimental evaluations provide useful insights into the effects of this integration, and answer the question whether this extended DBMS delivers increased performance.

1.3 Methodology

The methodological approach of this thesis is composed of the following elements:

1. **Literature Review**
   To fully understand the topics of structural query decomposition, Yannakakis' algorithm, join optimization and NewSQL databases, a thorough literature search, as well as review of the relevant literature, and discussion of the respective topics is carried out. This acts as the theoretical base of the thesis.

2. **Integration of the Optimization into TiDB**
   The identified optimization, including Yannakakis' algorithm [Yan81], is architecturally designed and integrated into TiDB and its query planner. Therefore, Yannakakis' algorithm is implemented and integrated into the query optimizer as an additional query optimization alternative inside the DBMS TiDB. This requires a substantial extension of the current query planner.

3. **Performance Evaluation of the System Including Yannakakis' Algorithm**

   a) **Experimental Performance Evaluation**
   The performance of the resulting system, compared to the basic system, is evaluated through a benchmark containing acyclic and almost acyclic queries. The data will be based on the MusicBrainz dataset, with queries from previous works of Mancini et al. [MKC+22] and Gottlob et al [GLL+23]. As the architectures and requirements of the databases used in [MKC+22] and [GLL+23] are different to those of TiDB, it is not useful for us to compare our results with theirs.

   b) **Analysis of the Results of the Benchmark**
   The results of the benchmark are analyzed as a foundation for the reasoning about the benefits and pitfalls of the newly integrated algorithm. In particular, there is also an analysis about which types of queries performed better or worse.

   c) **Interpretation and Discussion of the Results**
   Finally, the results are interpreted and discussed, whether they reflect the expected theoretic results, and what the reasons might be. A final conclusion is drawn regarding the usefulness of integrating Yannakakis' algorithm into the DBMS.

As query optimizers are complex and highly efficient software, the biggest challenge of the work is to integrate structure guided query optimization as additional optimization alternative in the optimal places within the query optimizer, and reuse as much code as possible. Additionally, the existing functionality and performance of the database should not be disrupted through this integration.

1.4 State of the Art

Current DBMS already implement heavy optimization techniques to boost query performance. This helps with a lot of problems, however, not so with the problem of exponential blowup during query computation.

As Chandra and Merlin [CM77] showed, the evaluation of conjunctive queries is NP-complete. Yannakakis [Yan81] was one of the first to identify that a subclass of conjunctive queries, the class of acyclic queries, is indeed solvable in polynomial time with respect to the input and output of the query, and provided an algorithm for the evaluation of such queries. A formal definition for acyclic and cyclic queries will be given in Section 3.1. As a conjunctive query and the respective scheme can be seen as a hypergraph, Fagin [Fag83] identified various classes of acyclic hypergraphs. Of these classes, alpha acyclic hypergraphs correspond to queries solvable by Yannakakis' algorithm [BB16]. Gottlob et al. [GLS02] generalized and extended the notion of acyclicity to almost acyclic conjunctive queries. This makes the methods with which acyclic queries can be efficiently evaluated applicable to almost all conjunctive queries. Furthermore, they found that hypertree decompositions with hypertree width 2 or 3 can be efficiently computed. This is sufficient, since most real-world queries are acyclic or almost acyclic, characterized by a hypertree width of at most 2 or 3 [FGLP21].

Following, the field of structural query optimization has seen some experimental systems and implementations, which integrated Yannakakis' algorithm:

Distributed processing: In the field of distributed query processing, Afrati et al. [AJR+17] proposed a Map-Reduce algorithm to compute a join query in a distributed way based on Yannakakis' algorithm and a generalized hypertree decomposition (GHD).

Security: Yannakakis' algorithm did also find its way into security research, as Wang and Yi [WY21] used the algorithm to evaluate queries across multiple parties, in a privacy preserving way, not revealing their local data to the other parties.

Graph databases: Concerning graph databases, Tu and Re [TR15] developed a new experimental query engine, DunceCap, which uses worst-case optimal joins, and Yannakakis' algorithm. This query engine was then used by Aberger et al. in EmptyHeaded [ALT+17], a relational high level, graph processing engine which uses Yannakakis' algorithm, among other optimizations and join algorithms, to efficiently process queries on graphs. Further, Aberger et al. [ALOR18] created LevelHeaded, a unified database engine, which combines abilities to compute Business Intelligence and Linear Algebra queries in order to be capable of handling machine learning algorithms. It is built on the same join processing techniques, including structural methods, as EmptyHeaded.

Queries under updates: Considering querying under updates, Idris et al. [IUV17] introduced a dynamic version of Yannakakis' algorithm, and developed a data structure which reduces the effort of IVM (Incremental View Maintenance), and replaces materialized results with this data structure, but still does not need complete recomputation of results on updates.

Relational databases: Finally, in relational databases, Ghionna et al. [GGGS07, GGS11] made an attempt to implement a query rewriter, that was experimentally also integrated in PostgreSQL, using hypertree decompositions and a variation of Yannakakis' algorithm (later named H-DB). This system works in a way that the root node (specifically the node's λ label) of the hypertree decomposition contains all relations needed for the output of the query. Therefore, in the root node of the decomposition, all relations directly involved in the output have to be joined, building all possible tuples for the output. This approach is suboptimal, since for the root of the decomposition, the complete join(s) have to be computed, possibly causing an explosion of intermediate results. However, their approach eliminates the second and third phase of Yannakakis' algorithm by directly joining along the structure of the decomposition, which is beneficial for some queries. More recently, Gottlob et al. [GLL+23] presented YanRe, a SQL query rewriter, which uses structure guided query evaluation and Yannakakis' algorithm to rewrite certain queries to a list of statements. These statements resemble the steps of Yannakakis' algorithm. They also identified a class of conjunctive queries, Zero-Materialization Answerable (0MA) queries, with special properties, enabling an efficient evaluation through rewriting.

Although the experimental systems and concepts mentioned above have shown very promising results, matching the possibilities which are theoretically possible through these algorithms, there is still no deep integration of a Yannakakis style query evaluation in a production database, and therefore no knowledge of the effects of such. This thesis targets a deep integration of Yannakakis' algorithm and structural query optimization into the NewSQL DBMS TiDB and evaluates the effects of this integration over a test dataset.

1.5 Results

Our contributions include the development and implementation of algorithms to integrate structure guided query optimization and evaluation deeply into existing database systems. Specifically, structure guided query optimization is integrated into TiDB (`https://github.com/pingcap/tidb`), which lies among the most popular Open-Source NewSQL DBMS. Our implementation is published on GitHub (`https://github.com/MichaelMartinek/StructureGuidedTiDB`). This integration is followed by a performance evaluation, which compares structure guided query optimization and evaluation with conventional query evaluation, and an analysis of the results.

The results of our evaluation show that some query instances benefit from our implementation of structure guided query optimization and evaluation. 40% of our full enumeration test queries, and 30% of our 0MA test queries were faster with our implementation. Furthermore, considering full enumeration queries whose result is not empty, our implementation dominates the original DBMS in a majority of instances. Although executing queries through Yannakakis' algorithm with our implementation increases memory usage to some extent, the outcome of our evaluation is promising towards a deep integration of these techniques into production databases, and suggests the usefulness thereof.

1.6 Structure of the Thesis

The remainder of the thesis is structured as follows: In Chapter 2, we discuss the relational data model, the basic data structure of Relational Database Management Systems (RDBMSs), outline challenges which appeared with bigger amounts of data, and look at a solution to scale relational databases towards big data with NewSQL, in particular TiDB. Following, we discuss how to solve the problem of exponential blowup for query evaluation from a theoretical viewpoint, in Chapter 3, including the introduction of 0MA queries, which allow for even more efficiency during query evaluation. Going from the theoretic view to the practical realization, we review the query evaluation workflow of TiDB, including references to the source code of the database system, and describe our implementation of structure guided query optimization into said DBMS in Chapter 4. Chapter 5 then contains evaluations of our implementation, as well as a discussion of the results. Finally, we conclude the thesis with Chapter 6.

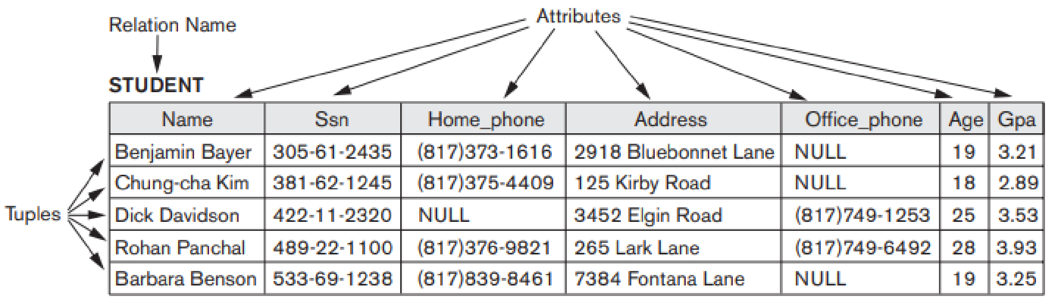
CHAPTER 2

Emergence of NewSQL

One of the predominant forms of data from a database perspective is relational data. From a high-level view, it is composed of entities, containing member variables, attributes. Entities are connected with other entities through relationships. This kind of data is queryable through a standardized query language, SQL. However, RDBMS, databases working with relational data and using SQL, are restricted when it comes to scalability and failure resistance, which could be realized by running multiple nodes in a cluster. This led to the rise of several types of NoSQL DBMS, which significantly improve those aspects. Furthermore, NoSQL offers a high flexibility regarding the structure of the data, often times without the necessity to define the data structure prior to using the database. Eventually, drawbacks regarding missing strict consistency guarantees, as provided by RDBMS, and a missing standardized query language brought the rise of another type of databases, NewSQL DBMS. NewSQL uses the relational data model, just as RDBMS, and is functionally equivalent to RDBMS in many aspects. Data storage is maintained by a cluster of key-value stores, which provides features as scalability and failure resistance. The actual computing nodes, however, are stateless, and can be scaled quickly. The NewSQL system we are considering in this thesis, TiDB, lies among the most popular Open-Source NewSQL DBMS.

2.1 Relational Data Model - RDBMS - SQL

The data model (or schema when talking about a specific instance) on which RDBMS are based is especially optimized at using minimal storage. It is characterized by the process of normalization and the underlying normal forms. Basically, in a high-level view, data saved in this data model consists of entities, containing member variables, attributes. Entities are connected with other entities through relationships. Relationships are constrained by their cardinalities, which indicate the number of concrete objects being linked by one specific relationship. Entities in the database are represented as relations, concrete instances as records, tuples, or rows of these relations. Records can be linked with other records of the same or other relations, using relationships through foreign keys, which basically reference other records. An example relation with labels is illustrated in Figure 2.1, an example relationship in Figure 2.2. [RG11]


Relation Name
Attributes
STUDENT
Tuples

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 2.1: Example Relation [EN11, Figure 3.1]

Manufacturer	
ID	Manufacturer Name
1	ACME
2	GoodCorp

Product		
ID	Product Name	Manufacturer ID
1	Paper	1
2	Pen	1
3	Ruler	2
4	Eraser	2

Figure 2.2: Example Foreign Key Relationship

A foundation in designing a relational data model are the normal forms, which provide several rules for the data structure in order to use the full potential of an RDBMS through avoiding redundancies and anomalies. The most common normal forms and their descriptions are listed in Table 2.1. These are for example the existence of a primary key, which fully and uniquely identifies a record and all its attributes in a relation. [EN11, RG11]

To fully understand the descriptions of the normal forms, the terms functional dependency and multivalued dependency have to be introduced:

A functional dependency is a tuple (X, Y), with the notation of $X \rightarrow Y$, where X and Y are attributes of a relation R, in which the value of X uniquely defines the value of Y. [EN11, Chapter 3.2.5]

A multivalued dependency (MVD) $X \twoheadrightarrow Y$, where X and Y are one or more attributes of a relation R, and two tuples t_1 and t_2 exist in the relation, has the following properties:

- Tuples t_3 and t_4 exist with:
- $t_3[X] = t_4[X] = t_2[X] = t_1[X]$

- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$
- $\exists Z,\, Z \in (R - (X \cup Y))$: $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$

If $Y \subset X$, or $X \cup Y = R$, then the multivalued dependency is trivial. [EN11, Chapter 15.6.1]

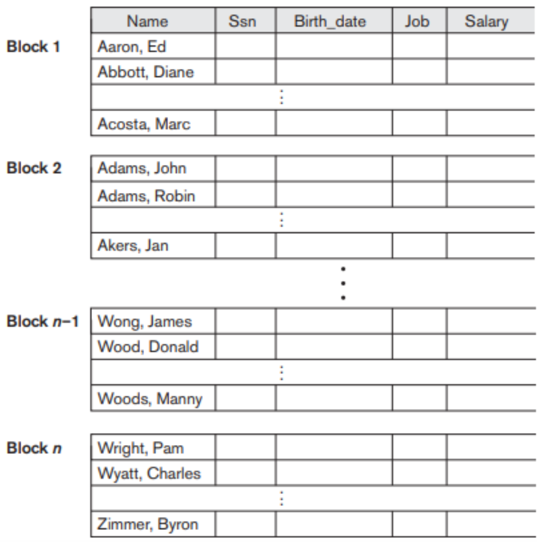
Normal Form	Description
1NF	All attributes of a relation are atomic, i.e., contain only one value, and relations are not nested.
2NF	If a primary key is composed of multiple attributes, all non-primary key attributes have to be functionally dependent of the whole primary key. Otherwise, all non-primary key attributes have to be functionally dependent on the primary key.
3NF	A non-primary key attribute should not be functionally dependent on another non-primary key attribute.
BCNF (Boyce-Codd)	An attribute of a key candidate (of which one or multiple other attributes are functionally dependent) should not be functionally dependent on a primary key attribute.
4NF	Multivalued dependencies ($X \twoheadrightarrow Y$) have to be trivial or X has to be a superkey (key candidate or a superset of a key candidate) of the relation.

Table 2.1: Most Common Normal Forms [EN11, Chapter 15.3 onwards]

A data model conforms to one of these normal forms, if all previous normal forms are conformed to, and the description of the current normal form is conformed to. In practice, data models are normalized until 3NF or BCNF. Higher normal forms are only used in special cases. There is also a 5NF, which is mostly irrelevant in practice and therefore omitted.

As previously mentioned, foreign keys implement relationships. These foreign keys are instances of other primary keys or attributes of records, which reference the said record from another record of the same or other relations. Furthermore, there are some restrictions, constraints, in RDBMS, which assure the structure and other properties of the data. The most popular among them are `UNIQUE`, which must be applied for primary keys, `PRIMARY KEY` itself, `FOREIGN KEY` referential constraints, `NOT NULL`, as well as user-defined `CHECK` constraints. [EN11, RG11]

The data is saved by the database in so-called database files. Each file contains a sequence of records, as illustrated in Figure 2.3, encoded in a certain format, which can be read and written performantly. Records can be saved unordered, which leads to a high write performance, with a worse read performance, and ordered according to certain criteria (e.g., numerical, alphabetical, or hashes of attributes), which takes longer to write, however the read performance, especially when specific entries are searched, is better. Storing and accessing the ordered records itself can be realized through multiple structures, with B or B^+ Trees often being used. Additionally, there are indexes. Primary indexes are realized as ordering of the relations database file(s), secondary indexes, as an additional index of a relation, are realized in separate database files, wherein every file, one secondary index can be contained. These files are structured in a key-value like format, having an ordered key field, which then has a pointer to the actual record in the database file. This pointer could for example be implemented through the value of the relation's primary index. [EN11, Chapter 17] [RG11, Chapter 8]



	Name	Ssn	Birth_date	Job	Salary
Block 1	Aaron, Ed				
	Abbott, Diane				
	⋮				
	Acosta, Marc				
Block 2	Adams, John				
	Adams, Robin				
	⋮				
	Akers, Jan				
	⋮				
Block *n*−1	Wong, James				
	Wood, Donald				
	⋮				
	Woods, Manny				
Block *n*	Wright, Pam				
	Wyatt, Charles				
	⋮				
	Zimmer, Byron				

Figure 2.3: Illustration of an Ordered Database File [EN11, Figure 17.7]

SQL is a standardized query language, which is supported by almost all RDBMS. SQL has sublanguages to query data (Data Query Language (DQL)), modify data (Data Manipulation Language (DML)) and define data structures, relations, and their relationships, in the database (Data Definition Language (DDL)). This standardized query language is one of the main reasons for the wide adoption of RDMBS. Another reason therefore is the level of consistency and concurrency delivered by these DBMS by using transactions, which enforce the ACID properties. The ACID properties are followed throughout the execution of a transaction. These are:

- **Atomicity**: Transactions are executed entirely, or not at all, representing an atomic unit of processing.
- **Consistency**: A transaction is executed as one uninterrupted unit of processing, meaning the state of the database after transaction execution is, just as before the execution, consistent, i.e., valid regarding to constraints.

- **Isolation**: A transaction appears to be executed as if it was the only running execution on the database.
- **Durability**: Changes in the databases state by finished/committed transactions are applied permanently, are not lost, and persist in the database.

Transactions may be successfully committed or aborted by the user or some error occurring during execution. [EN11, Chapter 21.3] [RG11, Chapter 16.1]

Isolation is available in multiple levels of varying strictness. These are in rising order:

- **Read Uncommitted**: All data written by other transactions, even uncommitted, can be read.
- **Read Committed**: Only committed data (even committed during the runtime of the current transaction) and changes done by the current transaction can be read. The data could be changed by other transaction during the transaction, which means there could be different values read for the same data.
- **Repeatable Read**: Only data committed before the current transaction started, and changes done by the current transaction can be read. If a value is read, any following read of this data gives the same value, as changes are not allowed.
- **Serializable**: Transactions appear as if they would be executed in sequential order.

Depending on which level of isolation is guaranteed or configured for an ACID transaction, different phenomena or anomalies can occur. These are caused by concurrent transactions and include dirty read (reading values written by other transactions which are rollbacked afterwards), non-repeatable reads (reading different values for the same data in a transaction) and phantom reads (reading more or less data, caused by other transaction inserting or deleting data) [ISO, RG11]. Most RDBMS feature isolation up to serializability. [CDD+23]

As a result of the properties guaranteed by ACID, the relational data model, and SQL, RDBMS are suitable for a broad variety of use cases, including a high number of concurrent users. Therefore, they have been the state-of-the-art solution for a lot of use cases.

2.2 Path to NewSQL

2.2.1 Challenges and Solutions

With the rise of Big Data, and new applications, which should have a high availability, have varying loads, and often produce non-uniform data, RDBMS could not satisfy all requirements anymore. RDBMS do have weaknesses when it comes to scaling[1] and failure tolerance, which is a reason why several types of NoSQL DBMS were developed. The most important types of NoSQL DBMS are key-value stores, document stores, column stores and graph databases. Most universally usable among these are document stores, which can save data of different structures in a distributed way to satisfy previously mentioned weaknesses of RDBMS. [Cat11]

NoSQL DBMS were developed with the goal of creating an easily scalable, highly available, failure tolerant and largely constraint free[2] DBMS. This is realized by running the database in a cluster consisting of multiple nodes. Each node stores a (overlapping) part of the data, and synchronizes changes in the state among them, to guarantee failure tolerance and availability. Furthermore, to scale the cluster, nodes can be quickly added or removed. [Cat11] The respective satisfiable property therefore is the elasticity [GHTC13]. The most important types of NoSQL DBMS for this thesis are the following:

- **Key-Value Stores**: Store data in a key-value structure, with arbitrary values for the key and the value field, similar to a dictionary. The key acts as an identifier.
- **Document Stores**: Similar to key-value stores, however, the DBMS understands the structure and the data of the value, also named document, typically a Javascript Object Notation (JSON) document, and is able to work with its contents. [Cat11]

In contrast to ACID, NoSQL DBMS mostly follow the BASE properties, which means the database is *BA*sically available, maintaining a *S*oft state and having *E*ventual consistent data. Therefore, the level of consistency is not as good as in RDBMS. [Cat11]
Although some NoSQL DBMS found solutions for the missing consistency guarantees and eventually also integrated ACID transactions (e.g., [Mon]), this still poses a problem for applications.

Besides, for distributed systems, which means also for the later mentioned NewSQL DBMS, the CAP theorem applies. This theorem claims that out of the three properties, *C*onsistency, *A*vailability and *P*artition tolerance, only two can be guaranteed. [Cat11]
Daniel Abadi further elaborated the challenges outlined through the CAP theorem, which does not distinguish between cases with a network partition, and normal operation, with the PACELC theorem, integrating these two states into the properties tradeoff. This theorem states that, if there exists a *P*artition in the network, there is the preference of either *A*vailability, or *C*onsistency, otherwise (*E*lse), in normal operation, the system design decides whether a low *L*atency, or *C*onsistency is preferred. This leads to the

[1]Scaling is the process of adding or removing (hardware) resources to/from a node or cluster. Vertical scaling describes the process of adding more resources to or removing resources from single nodes (also named scale up for more resources, and scale down for less resources). Horizontal scaling describes adding or removing whole nodes from a cluster (also named scale out for more nodes, and scale in for less nodes). When we mention scaling without further specification, we always mean horizontal scaling.

[2]Constraint free in this context means free of various database constraints, like FOREIGN KEY, UNIQUE, and CHECK.

result that systems delivering strict ACID properties, including serializability, have a PC/EC classification, preferring consistency in both cases, whereas more lax systems, like the previously discussed NoSQL systems with enhanced consistency (supporting ACID transactions) fall into the PA/EC category. NoSQL systems not focussing on consistency are categorized as PA/EL, preferring a low latency in normal operation, and availability in case of a network partition. [Aba12]

Additionally, NoSQL DBMS do not use a standardized query language or interface, which results in separate query languages and interfaces for each NoSQL system, mainly focusing on the basic Create/Read/Update/Delete (CRUD) operations. [Cat11]

2.2.2 NewSQL as Remedy

As a result of these pitfalls, another category of databases, NewSQL, was developed, which combines the features of RDBMS, including SQL, the relational data model and the consistency and concurrency, and NoSQL DBMS, with its abilities for scaling, failure tolerance and availability. NewSQL DBMS are multi-node DBMS, run in a cluster, which provide a separation of computing and storage, each in separate nodes. Nodes do not share any resources with other nodes, meaning they are designed as shared-nothing nodes. Optimally, the database is able to run on commodity hardware without the need for special real-time and time synchronization hardware. Storage nodes provide storage for the data state, whereas computing, or database, nodes are stateless, access the storage nodes through an API, and compute the data. This model enables fast and easy scaling, which describes the capability to realize elasticity and failure tolerance of the system. An example NewSQL cluster with computing/database nodes, storage nodes and their communication paths is illustrated in Figure 2.4. [OV20, Chapter 11.6.2]

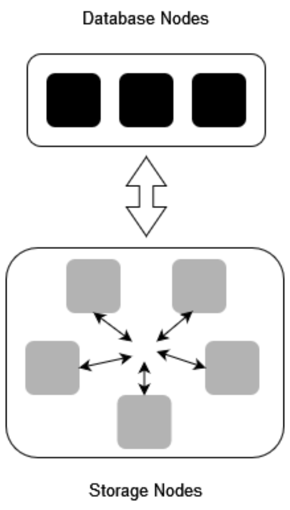


Figure 2.4: Illustration of a NewSQL Cluster with Database and Storage Nodes

Storage nodes synchronize the data among themselves (replication[3]) and divide data accesses to provide a better overall performance (partitioning[4] the data). Therefore, consensus algorithms, like Raft [OO14] or Paxos [Lam19], are used by these nodes to keep the state synchronized and consistent. This could, dependent on the used protocol, be done through synchronization with a master node, or through gossip between the nodes. Furthermore, replication and partitioning deliver availability and therefore failure tolerance, and scalability, for the storage.

Database nodes on the other hand are stateless and do not need to store any data. Their main job is to receive SQL statements, build an execution plan, and execute the plan by translating single relational operations into read/write instructions understandable by the storage nodes. In the course of execution, they have to keep the ACID properties for the transaction, finish computing the statement or transaction by combining the results received by the storage nodes, and return the result to the user. Thus, computing nodes contain multiple layers to realize the described flow. Generally seen, there is a SQL layer, a transaction layer, and a storage layer. The SQL layer handles the compatibility of the DBMS with SQL and executes queries on a high-level, building and optimizing the query plan, and executing it, including communication with lower layers. The transaction layer contains the adherence to concurrency and consistency guarantees, as for example the ACID properties, for data read from and written to the storage nodes. This typically involves checking or providing timestamps and versioning of storage entries. Finally, the storage layer manages the communication with the storage nodes, and executes SQL statements, or parts thereof, in a low-level view by issuing commands to read and write data from and to the storage nodes. Figure 2.5 illustrates the operations of the layers of a computing/database node.

As NewSQL databases are distributed systems, the level of isolation is slightly lower than with RDBMS. Whereas almost all RDBMS are able to deliver serializability, some NewSQL DBMS do not reach this level of isolation. [CDD$^+$23]

Other than RDBMS, NewSQL DBMS not only focus on Online Transaction Processing (OLTP) or Online Analytical Processing (OLAP) workloads. An important class of NewSQL databases support Hybrid Transaction/Analytical Processing (HTAP) workloads, which is a combination of OLTP with transactional workloads, and OLAP, with analytical workloads. [OV20] This also brings the ability to completely abolish the need for extra databases as data warehouse, and Extract/Transform/Load Operations (ETLs) to merge the data, as it would be needed with a combined SQL and NoSQL system. [VJPO21] Some NewSQL DBMS offer the functionality to connect and interact with them, additionally to the traditional SQL interface, through other interfaces, like Apache Spark [Pin23n],

[3]Replication is the process of synchronizing data across multiple independent nodes, which contain so called replicas, copies of the data. [CHMA21]

[4]Partitioning describes the process of splitting data into multiple parts. For vertical partitioning, data is split up into sets of different columns or attributes. Horizontal partitioning however splits data into sets of different rows or instances. An alternative name for this is sharding, in which the shards, or partitions, are split up across multiple storage nodes. When mentioning partitioning without the addition of horizontal or vertical, we are talking about horizontal partitioning. [CHMA21]

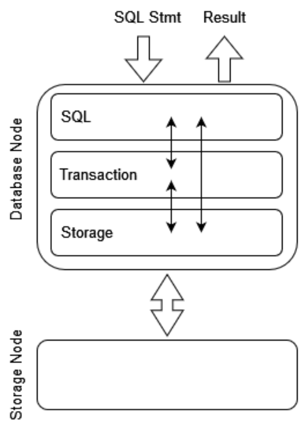


Figure 2.5: Illustration of the Described Layers

direct key-value interaction through some API [CHMA21, Chapter 4.3], or synchronize the data with other well known and often used platforms, e.g., Apache Kafka [Pin23a]. The aforementioned functionalities further enhance the analytical capabilities of HTAP NewSQL DBMS.

Examples of NewSQL DBMS are CockroachDB [Coc23] and TiDB [Pin23b]. We are going to extend TiDB in this thesis, therefore, we are taking a closer look at this database.

2.3 TiDB

TiDB (Ti comes from titanium and stands for the reliability of the database) is a NewSQL HTAP DBMS, which features a complete database solution including possibilities to interface with and integrate other data processing environments and technologies, manage a cluster and the possibility of a serverless cloud database. It has multiple components, some of which are mandatory, and some are optional. The basic architecture of TiDB consists of three component groups: TiDB, which is the database computing engine, a distributed storage layer which consists of two types of storage systems (one of which is mandatory), and a managing component, named Placement Driver (PD). An illustration of a basic deployment is shown in Figure 2.6. Responsibilities of these components, as well as other optional components and tools are listed below:

- **TiDB**: This is the main SQL and database computation engine. It receives statements from clients, manages transactions and communicates with other components in order to read or write data (storage layer), or to obtain timestamps and information about data distribution (PD).
- **PD**: The Placement Driver is a distributed component, that acts as the time oracle in the cluster, assigning monotonically increasing timestamps to transactions, and managing the placement of data on the storage cluster.

- **Storage**: The distributed storage consists of two storage engines, one of which has to be used.
  - **TiKV**: TiKV is a row-based distributed key-value storage engine. It is the main storage engine, as row-based data is frequently used with SQL. Internally, RocksDB is used as the key-value engine.
  - **TiFlash**: TiFlash is a column-based distributed storage engine. It is essential in answering OLAP Queries, as they can be computed more performantly in a column-based style. Furthermore, isolation between OLAP and OLTP, regarding the resource usage, is guaranteed through this component. Usage of this storage engine has to be explicitly configured for every relation of the data model.
- **TiSpark**: TiSpark enables to use the database with Apache Spark. Data can be used in a Spark environment through this component, as well as data can be read from other sources and inserted into TiDB. This component is an optional extension of TiDB and further enhances the OLAP capabilities of TiDB. [Pin23n]
- **TiCDC**: TiCDC (CDC = *C*hange *D*ata *C*apture) is a data migration tool, which directly interacts with the storage and PD in order to replicate data to other TiDB databases and export data to other platforms, like a MySQL Database or an Apache Kafka cluster. This is an optional tool. Data import in a very large scale is also possible with TiDB Lightning. [Pin23a]
- **TiUP**: TiUP enables easy deployment and maintenance of TiDB clusters, acting as a package manager for the cluster. This tool is optional. TiDB can also be deployed and managed in Kubernetes by TiDB Operator. [Pin23o]

2.3.1 Consistency

TiDB provides ACID transactions with isolation level Repeatable Read (RR), or Snapshot Isolation (SI), and Read Committed [HLC+20, Pin23m]. SI is a term, which is not strictly defined in the SQL standard. Some DBMS see SI as RR, some saw it equivalent to serializable. TiDB implements SI as RR. Additionally, optimistic, and pessimistic transaction modes are supported. This describes the point in a transaction when a lock of an object is obtained. With optimistic mode, locks are obtained before committing, in pessimistic mode, locks are obtained when data is written (for updates of data). When keeping the architecture of the database in mind, data is always written in the local memory of the database engine first, which means as soon as data is changed locally in TiDB, a lock is acquired in TiKV (TiFlash does not directly accept writes from TiDB). [HLC+20]

The transactional properties are enforced by all components of the cluster in a collaborative manner. TiDB, the computational engine, is the main transaction coordinator (for one transaction), and enforces the Two-Phase Commit (2PC) protocol. PD acts as the time

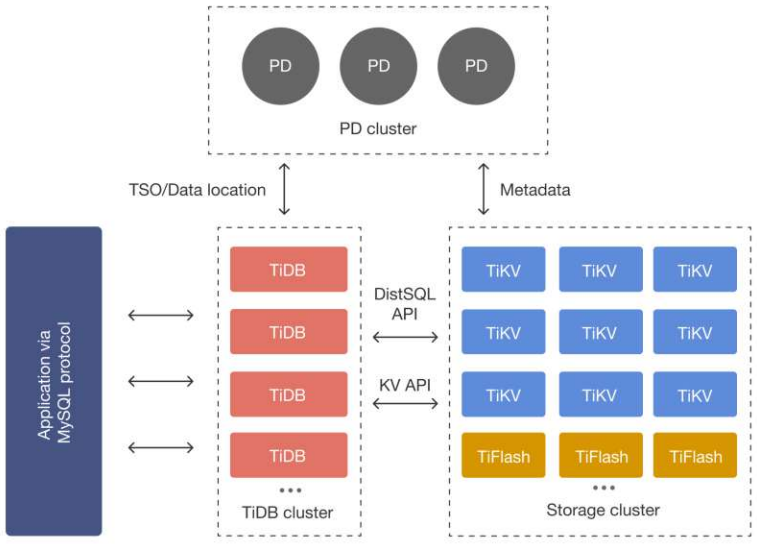


Figure 2.6: TiDB Architecture of a Basic Cluster [Pin23b]

oracle of the cluster, and TiKV provides the Multi Version Concurrency Control (MVCC) versioned datastore. Therefore, when TiDB receives a transaction begin, TiDB obtains a timestamp from PD, and reads the respective needed data for the current snapshot time/timestamp from TiKV. Afterwards, data manipulation, i.e., writes, inserts, or deletes, is carried out locally on TiDB. At commit, the 2PC is started. The changed data is locked on the storage (in case of a pessimistic transaction, this happens directly after local data manipulation on TiDB), and the changes are sent to the storage. In the case of successfully applying these locks and changes, a commit timestamp is obtained from the PD, and TiDB sends the commit to the storage. This concludes the 2PC. Finally, the result is returned to the client and secondary indexes are updated, as well as resources are released, asynchronously, by TiDB. [HLC+20]

2.3.2 Storage

TiKV stores data, other than RDBMS, in key-value pairs, which are part of a sorted map. The key of a pair, respectively a database record, is built as string similar to the following: `table[table_ID]_record[row_ID]`, and the value consists of the values of the record's columns. The `table_ID` is an internal identification for the current table, the `row_ID` is the value of the primary key, which is either a specifically defined primary key, or an internal artificial rowid. Secondary indexes have to be persisted separately, as in RDBMS. Depending if a secondary index is unique, or not, there are different storage formats. For a unique secondary index, the key consists of this format: `table[table_ID]_index[index_ID]_[indexValue]`, with the value of the primary key, respectively the rowid, being the value. For non-unique indexes, the key is `table[table_ID]_index[index_ID]_[indexValue]_[row_ID]`, with a null value. In both situations, for each record of the table participating in the index, one key-value pair in the mentioned format is required for the index. Since TiKV implements versioning through MVCC, version numbers, i.e., timestamps, are also included in the keys, such that for every key, multiple versions exist. [Pin23k, Pin23d]

The key space of TiKV is split in contiguous ranges of key-value pairs, so-called Regions. Every Region has a max size of 96 MB, and is replicated three times on the storage cluster. Additionally, for every Region, a Raft group, which is an instance of a Raft process, exists. [HLC$^+$20, Pin23k]

TiFlash on the other hand, stores columnar data. Therefore, a specialized data structure and storage engine, DeltaTree, is used to efficiently append current data in memory and apply the changes to a stable space on persistent storage afterwards. [HLC$^+$20]

Replication

The data on the storage has to be available, synchronized, replicated, and held consistent. This is done through the Raft [OO14] consensus algorithm [HLC$^+$20, CHMA21, Pin23k]. A Raft instance features three different node states, which are leader, follower, and candidate. A leader is the main node for a Raft instance, managing a Region in TiKV. There is only one leader per Raft instance, elected by all followers. It sends heartbeats, as AppendEntries message, to all followers, signaling regular operation of the leader and the whole Raft instance. Follower is the default state for every starting Raft participant. Other than the leader, which serves read and write requests, the followers are passive and serve as failure tolerance measure. If any of the followers do not receive a heartbeat in a certain time frame (e.g., because a node failed or crashed), it can start an election by sending a RequestVote message. The candidate role is used during the election of a new leader. All nodes, which do not get enough votes during an election fall back to the follower role. Raft splits the time in so-called terms, which describes a phase between two elections. The term is a monotonically increasing number, incremented for every new election. [OO14]

The basic mode of operation for a Raft instance is that the leader receives read and write commands from a client, in the case of TiDB, the database engine. This leader then creates logs from these commands, representing changes to the data state. Changes are replicated by sending them to the followers as AppendEntries message, which reply whether the new log is current for them and is appended to their log. They then also apply the new log entries to their local data state. [OO14] These two steps, locally appending the log message on the leader, and sending the new log to all followers, happen in parallel in TiDB, to optimize the process. Moreover, logs are sent to the followers in batches. [HLC$^+$20] Once the leader has received a majority or quorum of successful replies, the log entry is applied to the leaders local state. Afterwards, the result of the command is returned to the client. Even after this, until all followers have replied with a success, the leader retries to send the AppendEntries message for the log. This is important for followers which experience a network partition and are rejoining the network at a later point, or for slow followers. As long as there is one such case, all logs which are not applied for all followers are replayed, such that an order is guaranteed, and they do not fall behind the other nodes. To keep track of the logs in a term, a log index, which is a monotonically increasing logical timestamp, is introduced. Every log entry is identified by one log index. Since the leader is the main node in normal Raft operation, heartbeats and log replication, which are both AppendEntries messages with different arguments, are sent by the leader in a push-based way. [OO14]

In the basic Raft implementation, read requests can only be served by leaders. This causes performance drawbacks when a Region has a high load with multiple read requests, or a big amount of data to be read. As a logical consequence of the majority of the followers holding the current state (from a leader's perspective), reads can also be served by followers, named follower reads. In this case however, followers serving read requests always have to check if they have already applied the latest log by sending their current log index to the leader and asking if this is the current version. If not, they have to wait until the synchronization up to the current version, served through the leader's current log index, is done. This optimization realized in TiKV provides a significantly improved level of parallelization, considering that every Region is replicated at least three times, increasing the number of read serving TiKV instances from one to three. [HLC$^+$20]

Until now, we have only mentioned TiKV nodes for the Raft process. To keep the data in TiFlash instances synchronized and consistent, an additional node state, the learner, is added. A learner does not participate in elections and majorities for data replication, and therefore only adds low computational cost to the Raft process. Learners are part of a Raft group and just receive the logs from the leader, in the same order as other Raft participants, to keep consistency between TiKV and TiFlash. As TiFlash is column-based, logs have to be transformed. In this course, a compaction and removal of redundant information, as well as finally the transformation of the row-based logs into column-based data, is carried out. This data is then saved into the previously mentioned DeltaTree data structure. For the transformation of the logs from row-based to column-based format, the TiFlash node needs to know the current schema. Since the schema is stored as data in TiKV, TiFlash has a schema syncer component, which periodically, or at mismatches during data transformation, requests the newest schema from TiKV. The current schema is then cached in TiFlash. [HLC$^+$20]
In case the amount of data to synchronize is too big, for example when a new node joins the Raft process, the initial synchronization is performed using a snapshot of the current data state. [HLC$^+$20]
When reading from a learner, as with follower reads, the learner has to check with the leader of the Raft instance, if it holds the latest version, respectively the version with the requested timestamp. [HLC$^+$20]

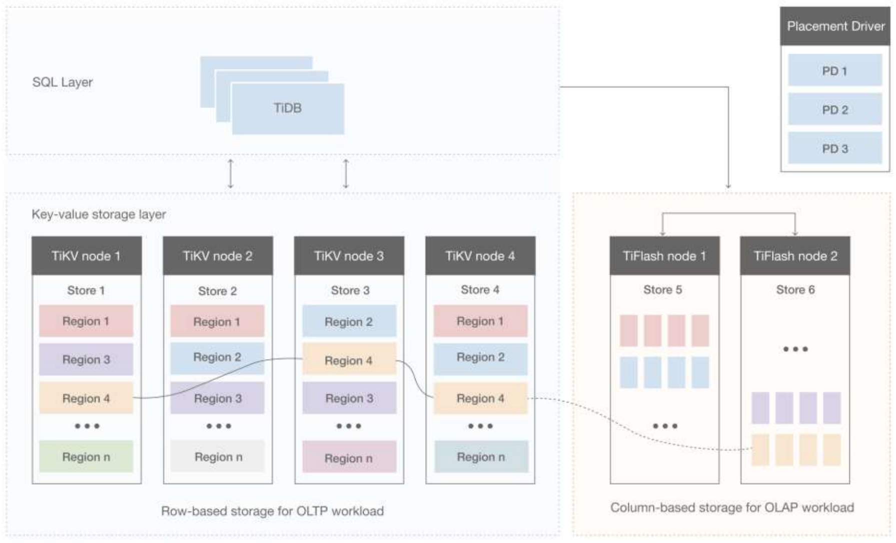


Figure 2.7: Multi-Raft System as Part of TiDB [Pin23k]

Balancing the Storage - PD

As already mentioned, every Region is managed by one Raft instance, applying the functionalities described above, making the cluster with multiple Regions a multi-Raft system. This is illustrated in the context of the architecture of TiDB in Figure 2.7. Regions with the same number or color symbolize a Raft group. To manage the participants of the Raft processes, distribution of Regions, and maintain the datastore, PD acts as a smart scheduling component, enforcing certain strategies to optimize the datastore. PD keeps track of distribution of data and usage on storage nodes, and keeps the balance in load and storage needs across the storage cluster. Finally, PD is the timestamp oracle in the TiDB cluster, and provides information on data location to TiDB. [Pin23c, Pin23h, HLC+20]

PD itself is operated as a cluster to ensure high availability and failure tolerance. This cluster normally consists of at least three members [Pin23c], which run the distributed key-value storage etcd[5], and therefore Raft, as a distributed datastore for a systems critical data, to keep data synchronized [TiK23]. The state of PD is not persistent, meaning the data PD needs is either collected entirely new from TiKV instances, or from other, already running PD nodes. [HLC+20] The collection of data from TiKV instances works through periodic heartbeat messages received from all TiKV nodes, containing information about hardware storage, storage performance, number of Regions residing on a TiKV node, and configured metadata regarding data placement (labels). Additionally, the heartbeat messages allow for deriving the availability state of the node. Moreover, leaders of a Region provide specific information about the Region. Along with inputs from the user, e.g., to scale the cluster in or out, decisions on how to schedule Regions

[5]https://github.com/etcd-io/etcd

are made. [Pin23h] Also, TiDB, the database engine, requests current metadata from PD, such as the physical location of data, i.e., the TiKV or TiFlash node, where a specific key or key range resides on the storage. This is important for the database engine to know which storage node to contact for reading or writing specific data. [HLC+20]

The strategies maintained by PD include the following:

- Keeping the correct number of replicas for every Region: E.g., if one storage node fails, move a new replica to another node.
- Distributing the replicas of a Region across the cluster: To keep the availability. E.g., all replicas on one storage node would be fatal in case of a failure.
- Distributing all replicas across the cluster: First step in balancing the storage need on the cluster.
- Distributing the leaders across the cluster: Since write operations have to be done through the leader of a Range, this helps to balance the load on the storage cluster.
- Distributing hotspots across the cluster: In case there are especially active Ranges, these have to be distributed in order to keep the performance high and balance the load.
- Balancing the storage need across the cluster: To keep one node from overflowing in case of sudden big write operations.
- Keeping the frequency of scheduling operations reasonable, according to configuration: Frequent scheduling operations could restrict the availability of the data for short periods of time. [Pin23h]

Following these constraints, the most important scheduling operations are adding a new replica to a Raft group, removing a replica from a Raft group, advise a Raft group to set a specific node as leader, and transfer Regions to other Raft groups [Pin23h], which involves splitting and merging Regions [HLC+20]. Note that only neighboring Regions can be merged, since they have to form one contiguous Range. The same applies to split operations. All the operations mentioned above besides merge and split can be realized through Raft only, and in one specific Raft group. For split, a respective Raft command is issued, which creates a new Region. Afterwards, a new Raft instance is started by PD on one of the storage nodes, which then synchronizes the data from the previous Raft group. Merge, however has to be controlled externally, since Raft does not provide the functionality to agree on killing one group and synchronizing its data to another one. [Pin23h, HLC+20] An example of a properly balanced storage cluster is illustrated in Figure 2.8. Region leaders are marked with an asterisk, hot Regions, meaning Regions with a high read/write load on this specific node, are marked with a red border.

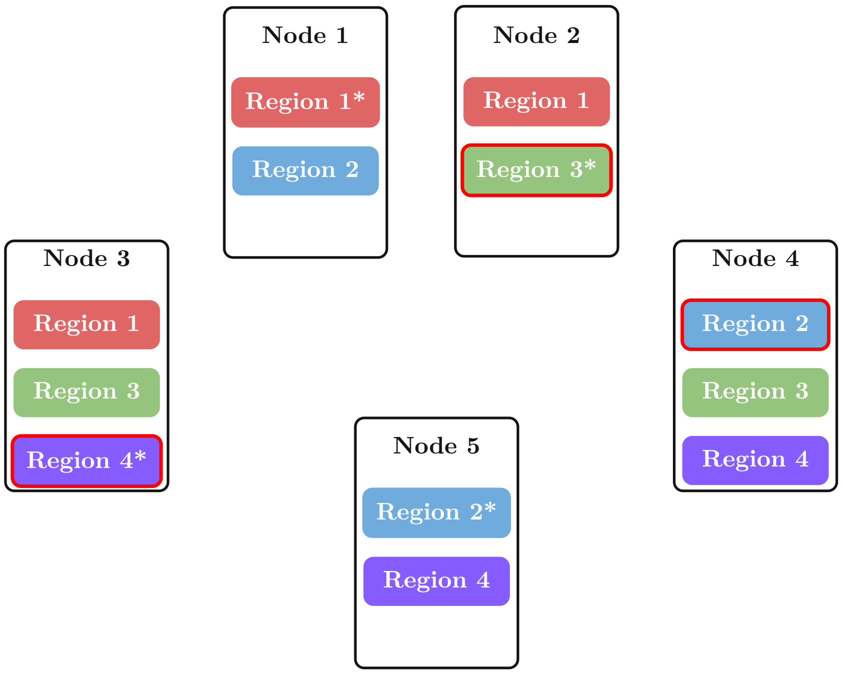


Figure 2.8: Balanced Storage Cluster

2.3.3 Query Planning and Execution

Finally, to utilize the previously described features, the actual SQL computation and execution engine is needed. TiDB supports the MySQL SQL dialect, and can therefore compute queries of this dialect. To begin with, the SQL statement has to be parsed into an Abstract Syntax Tree (AST). Based on this tree, a query plan realizing the semantics described in the statement is built and optimized. The optimization is the key process in enabling efficient and performant query execution. This optimized plan contains nodes (or operators), which can either be executed locally on the computation engine, or involve fetching some data from the storage nodes. The plan is executed in an iterator style execution, implementing the Volcano iterator model [Gra94]. Finally, the result is returned to the client. [HLC+20, Pin23d]

The query optimizer is composed of two parts. First, a RBO (rule-based optimizer), or logical optimizer, creates a logical plan from the AST. This logical plan is then optimized using certain rules which are always valid, independent from the dataset. This involves e.g., logical tautologies, changing the positions of certain operators, like selection and projection pushdowns, and removing redundant operators. The logical plan is then handed to the physical optimizer, a CBO (cost-based optimizer), whose task is to get the best physical plan, i.e., having the lowest costs, based on the actual dataset, cost for reading data, computing certain operators, etc. Finally, the optimized physical plan is executed. [HLC+20]

In TiDB, due to the distributed architecture of the database and storage, it is possible for the database engine to outsource certain operations to the storage cluster. Therefore, executions can be parallelized up to a certain degree. Normally, in non-distributed databases, operators can only be parallelized locally on the database engine. Thus, e.g., the hash bucket computation of hash joins can be sped up, or parallel branches of the execution plan can be executed concurrently. In TiDB however, some operations can be evaluated by a coprocessor implemented in TiKV or TiFlash, distributed on the cluster. The benefit here is that additional to the local parallelization on the database node, the storage nodes in which the data resides, are available for distributed parallelized execution, which then return the precomputed result to TiDB. Supported operations are for example selections using various predicates and functions, and some aggregation operations. This does not only bring the benefits of parallelization, it does also reduce the network usage. [HLC+20]

CHAPTER 3

Structure Guided Query Optimization

Querying describes finding data with a specified format and properties, as well as conditions. The DBMS has to search its data and find respective tuples meeting the mentioned conditions. Then, a result set can be returned. Queries specify the properties for the output data. However, they do not specify how to compute the result. This represents the declarative nature of SQL queries. The database has to create a so-called query plan, which specifies with which order of operations the result set can be obtained. There can be many query plans for a query, returning an equivalent result, however, possibly having wildly varying execution (and planning) times. Query optimization is the process of finding an optimal, or at least good, query plan for a query.

Query optimization is based on multiple steps, i.e., building an initial plan (modifying queries in this step is often referred to as query rewriting), followed by general, logical optimization, valid for every possible dataset (executed by a rule-based optimizer), and optimization specialized on the data in the database, physical optimization (executed by a cost-based optimizer).

Optimizers often used, like a traditional System R style optimizer [SAC$^+$79], Volcano [GM93], or a Cascades style optimizer [Gra95], do not use the full optimization potential. An initial query plan is just built recursively before handed to the logical optimizer, without further special meaning or optimization in this step. They then focus on transformations and search of the optimal access path. However, there would be information in the query itself which can be exploited for a possibly more performant execution of the query.

One of the most expensive operations when executing a query, joins, are reordered by these traditional optimizers, based on costs of different possible plans, discovered in the search space of the plan optimization process. The join order is the order in which (pairs) of relations and intermediate results are joined together. The goal of reordering joins is to get the smallest intermediate results, according to statistics from the database, which, in an optimal case, yields a low computational cost (of course also depending on the chosen implementations for operators). However, the information mentioned above, structural information given by the query itself, is not considered, although incorporating it in the query execution would solve the join ordering problem as a natural consequence.

Additionally, and with a potentially much bigger influence at the query execution time, the size of intermediate results would be reduced through structural methods, such that an exponential blowup of intermediate results, as it is possible without using these methods, cannot happen anymore. The tuples eliminated from the intermediate results are redundant, i.e., not contributing to the end result of the query. This would help especially when computing queries with a lot of joins, as the size of intermediate results can grow exponentially through many joins.

Structure based optimization focuses on the structure of the query first and tries to find an optimal query plan, which is based on the structure of the joins and their conditions in the query. Through building a hypergraph, decomposing this graph into a hypertree decomposition, and executing the query using Yannakakis' algorithm, exponential blowups of intermediate query results can be avoided. With this technique, every tuple in the intermediate results directly contributes to the end result of the query. Structure based optimization can be applied to queries being acyclic or almost acyclic, which makes up a huge part of all real-world queries, according to a recent analysis [FGLP21].

3.1 Queries

Queries are a declarative way to search for data in a database. As previously mentioned, they define a format and properties or conditions to which data has to conform in order to be included in the result set of the query. When operating a RDBMS, queries are formulated using the DQL sublanguage of SQL. SQL acts as a high-level representation of queries. Generally seen, queries for relational databases or data models can also be defined using relational algebra, logical expressions, or Datalog, which is another query language based on logical predicates, with a syntax similar to logical formulae.

Arguably, the most fundamental form of queries are conjunctive queries. Conjunctive queries are logically formulated, using existential $\exists$ operators and conjunctions of conditions, that is, conditions concatenated using the $\wedge$ operator, in first-order formulae [FGLP21, GLS02]. The formulae are built using atoms, representing relations. Atoms contain at least one parameter, as every relation is expected to have at least one column [GGGS07]. In these atoms, equality conditions for columns/attributes, as well as equi-join conditions can be embedded. Joins are expressed using the same variable in multiple atoms, defining an equality for an equi-join, i.e., a join that only uses equality conditions. As SQL statement, conjunctive queries would be of the structure *select ... from ... where ...* in which the select list defines the attributes of atoms to select, from defines from which data sources, i.e., relations or atoms, the tuples should origin, and where defines the conditions for the result tuples [GGGS07, GLS02]. From now on, in this chapter, when we mention queries, we mean conjunctive queries, as long as not declared differently.

As an example conjunctive query, we introduce a database schema (from [GLS02]) with the following relations and attributes:

```
enrolled(StudentPersID, CourseID, Date)
teaches(TeacherPersID, CourseID, Assigned)
parent(ParentPersID, ChildPersID)
```

The following example query selects all parents who are teachers, and whose child is enrolled to some course:

$$Q_{CQ}: \; ans(P) \longleftarrow enrolled(S, C', R) \wedge teaches(P, C, A) \wedge parent(P, S)$$

To specify a variable, i.e., set a (equality) condition, for example the ID of the teacher, the respective variable would be exchanged with the ID of the teacher.

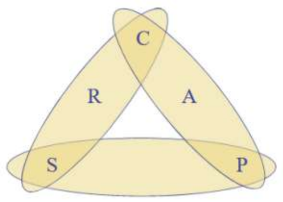
The left side of the query is named the head defining the queries result attributes/schema and the name of the resulting returned table, the right side is called body, defining the conditions for the query, and from which relations the data originates. Atoms of a query Q are denoted with $atoms(Q)$, attributes or variables of Q as $var(Q)$, and variables of a specific atom a as $var(a)$. [GGGS07]

Conjunctive queries return either concrete data, or a statement about the existence of data matching the query. The latter is also named a boolean conjunctive query, or existential query. A boolean conjunctive query is a conjunctive query just querying whether data matching the query exists, or not [GLS02]. An example for such a query, applying the same conditions as with the previous example is listed below. This query checks if there exists at least one parent which teaches a course, and their child is enrolled to some course.

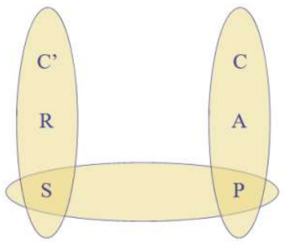
$$Q_{BCQ}: \; ans \longleftarrow enrolled(S, C', R) \wedge teaches(P, C, A) \wedge parent(P, S)$$

Chandra and Merlin showed that the evaluation of conjunctive queries is NP-complete [CM77]. However, there are some exceptions from this. Yannakakis' was one of the first to identify that a subclass of conjunctive queries, acyclic conjunctive queries, is indeed solvable in polynomial time with respect to the input and output of the query and provided an algorithm for the evaluation of such queries [Yan81]. This algorithm will be discussed further into the chapter.

Acyclic queries do not contain any loops regarding joins, i.e., with the overlap of variables of atoms. An example to differentiate acyclic and cyclic queries is shown in Figure 3.1. The circles around the variables represent atoms, the letters variables. In the query



(a) Cyclic Query
$enrolled(S, C, R) \wedge teaches(P, C, A) \wedge parent(P, S)$



(b) Acyclic Query
$enrolled(S, C', R) \wedge teaches(P, C, A) \wedge parent(P, S)$

Figure 3.1: Cyclic and Acyclic Query

illustrated in Figure 3.1a, a student who is enrolled in a course taught by their parent is queried, whereas in the query in Figure 3.1b a student, whose parent is a teacher, enrolled in any course is queried. That said, the former, as cyclic query, contains a circle. In the latter query, this circle is cut by introducing an additional variable C'. However, the acyclic query does not have the same meaning as the cyclic query.

As a proof of the usability of the polynomial evaluation of acyclic conjunctive queries, Fischl et al. [FGLP21] analyzed hundreds of real-world and benchmark database queries for their acyclicity. The result of the analysis found that most real-world database queries are acyclic, or at least almost acyclic, a notion that will be handled further into the chapter, and also holds beneficial properties of an acyclic query. Therefore, the polynomial time evaluation is applicable to these queries.

Formally, a query is acyclic, if there exists a join tree for the query. As the name join tree already implies, no circles are allowed. They essentially define an efficient order, i.e., a query plan, in which a query can be evaluated. A join tree $JT = \{V, E\}$ of a (conjunctive) query Q consists of vertices $V = \{v \mid v \in atoms(Q)\}$ and edges $E \subseteq \{(a, b) \mid a, b \in V \wedge var(a) \cap var(b) \neq \emptyset\}$. This means, vertices are labeled with their atom, e.g. $Parent(P, S)$, and edges connect two vertices having a variable intersection in their atoms. The latter is not a formal necessity, however, omitting the variable overlap would introduce cartesian products in the join tree. Following this definition, for every two vertices $a, b \in V$ containing equal variable(s), there exists a path from a to b. The special property of the path is that all vertices on the path between a and b also contain the same variable(s). Additionally, join trees meet the so-called connectedness condition. Thus, in a join tree, vertices with matching variables form a connected subtree. [GLS02]

In Figure 3.2, we illustrate an example join tree. This join tree corresponds to the query Q_{CQ} on page 27. The connectedness condition can be observed, on the root and the left child, which contain the variable P, and the root and the right child, which contain the variable S. There are no variables which do not form a connected subtree. Furthermore, as we can see, the cyclic query illustrated in Figure 3.1a has no join tree, since there exists a circle.

Note that for a query, multiple join trees can exist. They could differ in which vertex is chosen as root. We could use each of the three vertices as root of our join tree in Figure 3.2, changing the shape of the tree. Additionally, the way which vertex is connected to which other vertex could be changed, as long as the mentioned conditions are adhered to.

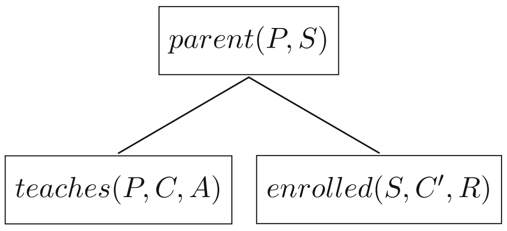


Figure 3.2: Join Tree of the Query Q_{CQ}

Join trees for queries can be efficiently computed (if existent), and therefore also their existence decided, with the GYO-reduction (Graham, Yu, and Ozsoyoglu) [Gra79, YO79]. The GYO-reduction itself checks whether there exists a join tree. Additionally, if the check decides that the query is acyclic, an order, including so-called witnesses, is created, which defines one possible join tree.

In order to understand the algorithm, we need to define two terms. An atom $r_1(\vec{Z_1})$ is contained in an atom $r_2(\vec{Z_2})$, if $\vec{Z_1} \subseteq \vec{Z_2}$. Furthermore, a witness $w(\vec{Z_1})$ for an atom $a(\vec{Z_2})$, is an atom, such that a only contains variables which do not occur in other atoms, and variables which are contained by the witness w.

The GYO-reduction algorithm [Gra79, YO79] can be described in multiple ways. We use an approach, which makes it easy to create a join tree afterwards. It is carried out as follows: First, atoms which only contain variables which do not occur in other atoms are eliminated and appended to the ordering. Next, all atoms for which witnesses exist, can be eliminated. The eliminated atoms are appended to the ordering, and the witnesses are remembered for building the join tree. This is repeated as long as there are atoms to eliminate.

If all atoms are eliminated at the end, a join tree, which can be built with the collected ordering of atoms, and the remembered witnesses, exists. The join tree is built as follows: In reverse order, take an atom from the ordering, and add it to the join forest, i.e., multiple join trees. The first one, and other atoms which were no witnesses, are roots. If an atom r_a was a witness of another atom r_b, then r_b is a child of r_a in the join tree. The result is a forest of join trees. They can be merged arbitrarily to form one possible join tree.

As an example for a cyclic query, we can take a look at the query in Figure 3.1a. There are no atoms containing only unique variables, as well as no atoms for which a witness exists. Each atom would span across two other atoms, enrolled contains variables of teaches and parent, parent of teaches and enrolled and teaches of enrolled and parent. Therefore, we cannot eliminate all atoms and have to decide that this query is not acyclic.

For an example for an acyclic query, we add two new relations, `grade(StudentPersID, CourseID, Grade)`, and `exam(StudentPersID, CourseID, Percentage)` to our schema, and consider the following query:

$$Q_{JT\_Ex}: \ ans(P) \longleftarrow enrolled(S, C', R) \wedge teaches(P, C, A) \wedge parent(P, S) \wedge grade(S, C', G) \wedge exam(S, C', P')$$

The query asks for a person who is a teacher and teaches some course. Additionally, the teacher's child is enrolled to some course, took an exam at this course and has a grade assigned for that course.

First, we check if there are atoms with unique variables. This is not the case. Then, we look for witnesses. The first witness is enrolled, which is a witness for grade, which we eliminate. Next, we eliminate exam, whose witness is again enrolled. The next step is to eliminate enrolled, having its witness in parent. Then we eliminate teaches. And finally, its witness, parent. We managed to eliminate all the atoms, consequently, Q_{JT_Ex} is an acyclic query.

Now we need to build the join tree. As described, we start with the last atom in the ordering, parent, which will be the root of the join tree. We then know that the next atom, teaches had parent as its witness, adding it as child of parent. The same happens with enrolled. Finally, enrolled was a witness for grade and exam, which will be added as children of enrolled. The finished join tree is illustrated in Figure 3.3.

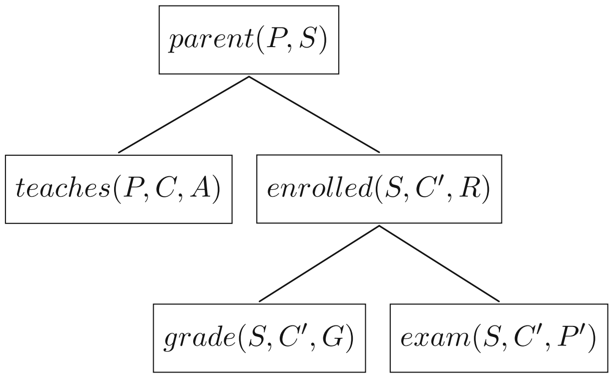


Figure 3.3: Join Tree of the Query Q_{JT_Ex}

3.2 Query Decomposition

Structural information of a query is extracted through decomposition of said query. For this, first, a hypergraph for the query has to be created. Following, the hypergraph has to be decomposed through a hypertree decomposition. In the course of creating a hypertree decomposition, the query is also checked for its degree of acyclicity, i.e., if the query is qualified to be executed with structural methods. The hypertree decomposition contains all the structural information necessary for query evaluation.

3.2.1 Query Hypergraph

A hypergraph $\mathcal{H} = (V, H)$ is a generalization of a graph in which, different to a normal graph, the edges, or hyperedges H, can connect multiple vertices at once. Theoretically, every hyperedge $h \in H$ contains of a set of vertices ($h \subseteq V$) which are connected by this hyperedge.

In the context of queries, we have already seen illustrations of hypergraphs in Figure 3.1. The circles represent atoms, or hyperedges. The variables are vertices of the hypergraph. Therefore, we define the query hypergraph of a query Q as follows: $V = var(Q)$, $H = \{\{v_1, ..., v_n\} \in V \mid atom(v_1) = ... = atom(v_n)\}$. The edges of $\mathcal{H}$ are also denoted as $edges(\mathcal{H})$, vertices or variables as $var(\mathcal{H})$. Every vertex connecting to more than one hyperedge represents a SQL join column. [SGL04]

As an additional example, we are going to build a hypergraph for the following query:

$$Q_{HG\_Ex}: \ ans \longleftarrow a(A, B) \wedge b(B, C) \wedge c(B, D, E) \wedge d(E, F)$$

The corresponding hypergraph in Figure 3.4 shows four circles for the atoms, respectively hyperedges, as well as the vertices. Vertex B connects to three hyperedges, E to two hyperedges. These two vertices match with the joined columns of a respective SQL statement.

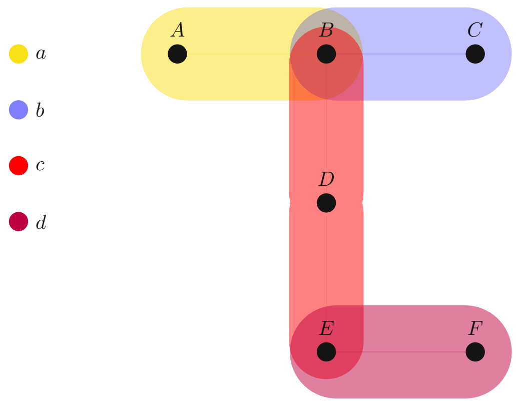


Figure 3.4: Query Hypergraph of the Query Q_{HG_Ex}

3.2.2 Hypertree Decompositions

A hypertree decomposition for a hypergraph $\mathcal{H}$ consists of a triple $\langle T, \chi, \lambda \rangle$ where $T = (N, E)$ is a rooted tree, and λ and χ are node labeling functions, which associate each vertex (or node) $p \in N$ with a set $\lambda(p) \subseteq edges(\mathcal{H})$, representing relations or atoms,

and a set $\chi(p) \subseteq var(\mathcal{H})$, representing attributes or variables. Instead of $\chi(p)$, we can as well write $var(p)$. $\lambda(p)$ is also named the cover (or edge cover, because it covers vertices of the respective hypergraph, forming edges) of p, $\chi(p)$ the bag of p. A subtree of T rooted at the vertex p is denoted as T_p. Furthermore, the width of the hypertree decomposition is defined as $max_{p \in N}|\lambda(p)|$, meaning the maximum number of relations in a cover across the hypertree decomposition. [SGL04, GGGS07, GLS02]
A hypertree decomposition with hypertree width 1 is acyclic, all with a width above 1 are cyclic. Fagin [Fag83] defined the term alpha acyclic for a hypergraph, which corresponds to an acyclic hypergraph. This is similar to an existing join tree for a query [GGLS16]. Therefore, alpha acyclic hypergraphs define queries solvable by Yannakakis' algorithm [BB16].

Continuing our thoughts of decomposing a query Q, by first creating a hypergraph $\mathcal{H}$, and then creating a hypertree decomposition $HT = \langle T, \chi, \lambda \rangle$ from the hypergraph, the following conditions are met by HT:

1. For each atom $a \in atoms(Q)$ of the query, (which created a hyperedge $h \in edges(\mathcal{H})$ in the hypergraph), there exists a $p \in vertices(T)$, such that $var(a) \subseteq \chi(p)$, i.e., the variables of every atom are covered by some bag.

2. For each variable $va \in var(Q)$ ($var(Q) = var(\mathcal{H}) = vertices(\mathcal{H})$), there exists a set $\{p \in vertices(T) \mid va \in \chi(p)\}$ in the hypertree decomposition, which forms a connected subtree of T, realizing the connectedness condition. Thus, every vertex of the connected subtree, and therefore every vertex on a path between two vertices containing the variable, also contains this variable.

3. For each vertex $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$, i.e., the attributes in the bag of p are part of the relations in the cover of p.

4. For each vertex $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$, with T_p being the subtree of T rooted at p. I.e., all attributes whose relations are in the cover of p and are contained in any bag of the subtree rooted at p are part of the bag of p. This is also named the special descendant condition. [SGL04, GGGS07, GLS02]

A hypertree decomposition $HT = \langle T, \chi, \lambda \rangle$ of a query Q is complete, if for each atom $a \in atoms(Q)$, there is a vertex $p \in vertices(T)$, such that $var(a) \subseteq \chi(p)$ and $a \in \lambda(p)$ [SGL04, GLS02]

In case there exists a complete hypertree decomposition for a query, which is acyclic, i.e., having a hypertree width of 1, efficient structural query evaluation methods can be used. The notion of completeness however, can be established easily in linear time, and in logspace, for every existing hypertree decomposition. Therefore, atoms not contained in a hypertree decomposition can be added as children of other atoms containing the same variables. This is possible, since for every atom, variables must occur in a vertices bag, which is described by condition 1. As a result, the size of the complete hypertree decomposition HT' of the query Q is $O(||HT|| + ||Q||)$. The width of the hypertree decomposition is not affected by the transformation. [GLS02]

Completeness is important for the evaluation to obtain the right result set, joined with every dependency. An incomplete hypertree decomposition would leave out some atoms, leading to a possibly incorrect result.

Gottlob et al. [GLS02] found that hypertree decompositions with a low fixed width of up to 2 or 3, are tractable. Contrary to the strict differentiation between acyclic and cyclic queries, there is also the notion of almost acyclic queries. These are characterized by queries for which hypertree decompositions with a low hypertree width of 2 or 3 exist. These almost acyclic hypertree decompositions resolve cyclic queries to almost acyclic queries. Due to the low hypertree width k, which is used as constant exponent for the complexity of the evaluation, $O(||D||^k * ||Q||)$ for boolean conjunctive queries, or $O((||D||^k + ||Out||) * ||Q||)$ for a full enumeration, where $||D||$ is the size of the database, $||Out||$ is the size of the query's output, and $||Q||$ is the size of the query, the evaluation is efficiently computable for a low k [GLL$^+$23]. Therefore, these queries can also be efficiently computed using structural methods.

As an empirical study by Fischl et al. [FGLP21] showed, this is sufficient for most queries. In a recent paper they analyzed hundreds of real-world and benchmark database queries for their acyclicity. The result of the analysis found that most real-world database queries are acyclic, or at least almost acyclic, which can be handled well by Yannakakis' algorithm.

A generalized form of hypertree decompositions, generalized hypertree decompositions (GHD), is defined just as hypertree decompositions, with the exception of the special descendant condition. They also deliver the same beneficial properties necessary for decomposition of a query and efficient structural query evaluation. [GGGS07, GLS01]

According to Gottlob et al. [GGLS16], only hypertree decompositions are generally suitable for decomposing and computing queries. This is the case, because they meet the following three conditions:

1. **Generalization of Acyclicity**: A certain fixed width k indicates an acyclic query.
2. **Tractable Recognizability**: The time complexity of recognizing queries of the fixed width k is polynomial.
3. **Tractable Query-Answering**: Answering of queries of fixed width k is possible in polynomial time.

GHDs generally do not satisfy the second condition, since for any fixed k ($k \geq 2$), it is NP-hard to decide if there exists a decomposition with width k [GGLS16, FGP18]. This is due to the missing special descendant condition, which restricts the search space in hypertree decompositions, but not in GHDs.

Fischl et al. [FGLP21] developed some new and improved algorithms, including BalSep, using balanced separators, creating sub-hypergraphs from input hypergraphs, to compute GHDs efficiently. These algorithms achieve a tractable generalized hypergraph decomposition with fixed width k through leveraging the so-called bounded intersection property and bounded multi-intersection property [FGP18]. These properties define favorable forms of hypergraphs, e.g. enabling tractable computation of a GHD for any fixed $k \geq 1$. The algorithms proposed in the paper are designed to deliver a result relatively quickly, similar to det-k-decomp [GS09]. This is opposing to opt-k-decomp [GLS99], which is designed to deliver an optimal hypertree decomposition, however, not being that efficient [GGLS16]. As a result thereof, the decompositions created might not be optimal, although having a certain width k, guaranteeing a certain degree of quality of the (generalized) hypertree decomposition.

As an example for a hypertree decomposition, we consider the following query:

$$\begin{aligned} Q_{HD\_Ex}: \; ans \longleftarrow & a(S,X,C,F) \wedge b(S,Y) \wedge c(C,Z) \wedge d(X,Z) \wedge e(Y,Z) \wedge \\ & f(F,G) \wedge j(J,X,Y) \end{aligned}$$

A corresponding hypertree decomposition for this query is illustrated in Figure 3.5. The resulting (G)HD has a width of 2. The first line in the nodes represents the edge cover λ of the node, the second line shows the bag χ of the respective node.

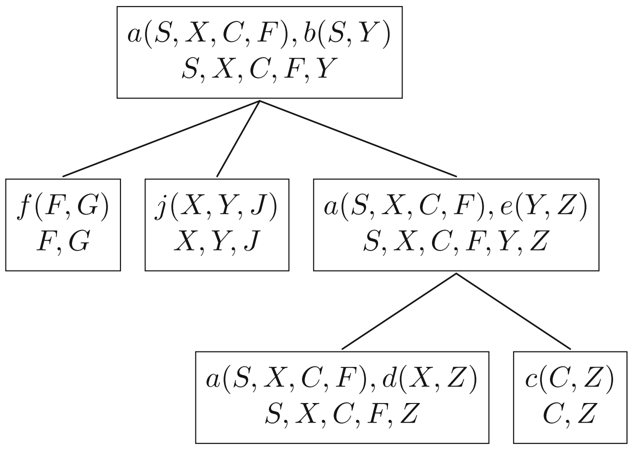


Figure 3.5: (G)HD of the Query Q_{HD_Ex}

The result of a hypertree-, or generalized hypertree decomposition, can be seen, similarly to a join tree, as a plan on how to evaluate a certain query. However, for relational databases, further steps have to be added to transform a hypertree decomposition into a real join tree, resulting in a query plan.

3.3 Yannakakis' Algorithm - Structure Guided Execution

Yannakakis' [Yan81] was one of the first to identify that a subclass of conjunctive queries, the class of acyclic queries, as we discussed before, is indeed solvable in polynomial time with respect to the input and output of the query, and provided an algorithm for the evaluation of such queries. Additionally to simple acyclic queries, almost acyclic queries can also be handled by this algorithm. This is due to the existence of a hypertree decomposition for these queries. In normal query execution, the size of intermediate results can exponentially grow, due to the execution of join operations on the whole dataset. In this approach, also tuples not contributing to the end result, which may be dropped by a later join, are contained in the intermediate results. Therefore, they grow, could easily exceed available memory and storage, and get exponentially big. As a consequence, the amount of data causes the evaluation of the query to either take very long, or even not finish at all in reasonable time.

The key operation enabling the strength of Yannakakis' algorithm when executing acyclic and almost acyclic queries, and preventing exponential growth of intermediate results, is the semi-join. A semi-join, denoted as $\ltimes$, is a join operation in which, as with normal joins, certain conditions should apply in order for a tuple to be part of the result. However, different to a normal join, a semi-join result only includes the attributes of one operand [VG84]. Additionally, each tuple of the included operand is at most contained once in the result, even if there were multiple matches. In Yannakakis' algorithm, we are only using equi semi-joins, relying on equality conditions.

The execution of the algorithm consists of the following steps:

- **Step 0**: This is a preparatory step, which prepares the base data for each node p of the hypertree decomposition. It is only necessary if the edge cover has a size of at least two $|\lambda(p)| \geq 2$. In this case, all relations of the edge cover are joined together, to form the base data of the node. The data is then used in the following steps. In case the size of the edge cover is 1, the base data is the data in the single relation. When we refer to data of a node in the next steps, we always mean either the data from the previous step(s), or the base data.

- **Step 1**: Bottom-up semi-join traversal: This step starts the elimination of dangling tuples, i.e., tuples not contributing to the end result. Therefore, it is checked whether hypertree decomposition nodes further down than the current one justify keeping tuples. In order to achieve this, we execute semi-joins $p_{new} = p \ltimes c$ in a bottom-up fashion, such that every parent p checks with each of its children c, whether to keep tuples, or drop them. If a node has multiple children, the computations are carried out sequentially. The data of each node (except leaves), is set to p_{new} after execution of the semi-join. This translates to the formula $R'_t = \pi_{var(t)}(\bowtie_{v \in V(T_t)} R_v)$ for each node $t \in T$, with R'_t being the data for the node t after this step, assuming that all attributes of a node's relations are contained in the bag of the node [GLL+23].

As previously mentioned, this step is sufficient for boolean conjunctive queries, as well as queries which can be answered through the root node of the hypertree decomposition. Because of the semi-joins, every tuple in the root node is backed by at least one tuple of every other node. Therefore, the root essentially contains a distinct enumeration of the result for the relations in its edge cover, which can be used to answer queries, as we will see further into this chapter.

- **Step 2**: Top-down semi-join traversal: For the second part of eliminating dangling tuples, we execute the same procedure in a top-down fashion, meaning we execute semi-joins $c_{new} = c \ltimes p$, setting the data of all children for the whole hypertree decomposition to their respective c_{new}. This happens for every node except the root, since it is not a child of any other node. Afterwards, we have computed the fully reduced dataset, represented by the formula $R''_t = \pi_{var(t)}(\bowtie_{v \in V(T)} R_v)$ for each node $t \in T$, with R''_t being the data for the node t after this step, and assuming that all attributes of a node's relations are contained in the bag of the node [GLL+23].
  If a join along the edges of the hypertree decomposition is conducted now, all tuples will find a match, hence directly contributing to the result of this join.

- **Step 3**: Bottom-up join traversal: The only step missing now is the computation of the final results. As previously mentioned, we join along the edges of the hypertree decomposition in a bottom-up manner, in order to calculate the complete solution. Since we performed a reduction in steps 1 and 2, all tuples have matches and thus directly contribute to the end result. Therefore, each parent p is joined with the result of its children c, $p_{res} = p \bowtie c$. The childrens' result is either their previous data (in case of a leaf node), or a result computed in this step. If a node has multiple children, the computations are carried out sequentially. Resulting from this step, every node contains all results of its subtree, represented by the formula $R'''_t = \pi_{var(T_t)}(\bowtie_{v \in V(T)} R_v)$ for each node $t \in T$, with R'''_t being the data for the node t after this step, and assuming that all attributes of a node's relations are contained in the bag of the node [GLL+23]. Consequently, the root node contains the full enumeration for the hypertree decomposition, and the query.

The join conditions are always determined by an intersection of the bags of the respective nodes, and the existence of the attributes/columns in the current schema. Note that instead of a hypertree decomposition, a join tree could be used. The edge cover of a hypertree decomposition node translates to the relation of a join tree node.

Although unusual for evaluating a query with a join tree and query plan in relational databases, this algorithm reuses intermediate results several times, in particular nodes with more than one child in the second step, as well as internal nodes in the third step. This is slightly different to the concept of the iterator execution model, which passes tuples directly to the next operator as they are computed.

The runtime of a query executed through Yannakakis' algorithm depends on the type of the query. If the query is a boolean conjunctive query, or can be answered by just using the relations in the edge cover of one node from the hypertree decomposition, the algorithm only has to be conducted until step 1 is finished. Therefore, the complexity is polynomial with respect to the input. If a full enumeration of the data is needed, all three steps of the algorithm have to be executed in order to answer the query. Following, the complexity is polynomial with respect to the input and output, as we described in Section 3.2.2 on page 33. Additionally, as mentioned earlier, the width of the hypertree decomposition has a big effect on the complexity. Every hypertree decomposition node's base data has to be built through joins. With the worst case of a cartesian product, the width directly affects the amount of data to be processed in later steps. Thus, the (constant) width k is used as exponent of the size r of the biggest input relation, which defines the input for one node as r^k [GGLS16]. However, this is not too much of a problem, since Fischl et al. [FGLP21] found that most real-world queries have a low hypertree width.

As an example, we are going to evaluate the following query with Yannakakis' algorithm:

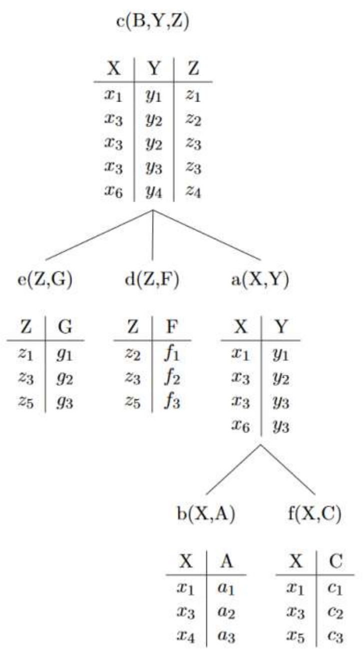
$$Q_{Yan\_Ex}:\ ans \longleftarrow a(X,Y) \wedge b(A,X) \wedge c(B,Y,Z) \wedge d(Z,F) \wedge e(Z,G) \wedge f(C,X)$$

The preparatory step can be skipped, because the width of the given hypertree decomposition is 1. An illustration of the hypertree decomposition, including the data in the relations is shown in Figure 3.6a.

The first step in Yannakakis' algorithm is to conduct a bottom-up semi-join traversal. Thus, tuples not backed by a child node are eliminated. Therefore, we start at the bottom, with a non-leaf node. This would be the relation $a(X,Y)$. We execute the semi-join $a \ltimes b \ltimes f$, to get the new data for a. The same is done for c with its children, including the result of the just mentioned semi-join for a. Eliminated tuples are marked with a strike through in Figure 3.6b, which displays the intermediate result after this step. Since in the next step, we do not change the root node, and its data is backed by all other nodes of the hypertree decomposition, the root would already be prepared for answering a query. This includes a boolean query, also named an existential query, which just asks if there is some data in the queries answer. In this case, the question can be answered with yes, since there are tuples left in the root node's data. Additionally, queries, which are just outputting data from variables contained in the root node, and do not need exact multiplicity, as we mention later, can be answered.

Next, the top-down semi-join traversal has to be carried out. In this step, previous results may be used several times. It is executed for all nodes, except the root. Therefore, we set the data of e.g. e as $e \ltimes c$, d as $d \ltimes c$, and a as $a \ltimes c$. As we can see, we used c three times in this computation. The same procedure is performed for the nodes in the next lower level. The resulting and fully eliminated data is illustrated in Figure 3.7a.

Finally, the actual joins can be computed in the last step. Every tuple has a join partner during this procedure. The structure of the hypertree decomposition acts as a join tree,



c(B,Y,Z)

X	Y	Z
x_1	y_1	z_1
x_3	y_2	z_2
x_3	y_2	z_3
x_3	y_3	z_3
x_6	y_4	z_4

e(Z,G)

Z	G
z_1	g_1
z_3	g_2
z_5	g_3

d(Z,F)

Z	F
z_2	f_1
z_3	f_2
z_5	f_3

a(X,Y)

X	Y
x_1	y_1
x_3	y_2
x_3	y_3
x_6	y_3

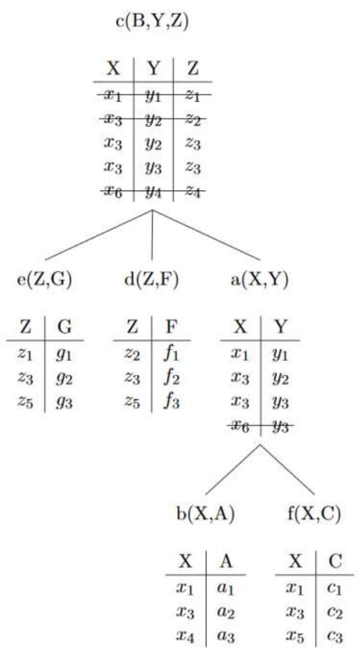
b(X,A)

X	A
x_1	a_1
x_3	a_2
x_4	a_3

f(X,C)

X	C
x_1	c_1
x_3	c_2
x_5	c_3

(a) (G)HD of the Query Q_{Yan_Ex}, with Data



c(B,Y,Z)

X	Y	Z
~~x_1~~	~~y_1~~	~~z_1~~
~~x_3~~	~~y_2~~	~~z_2~~
x_3	y_2	z_3
x_3	y_3	z_3
~~x_6~~	~~y_4~~	~~z_4~~

e(Z,G)

Z	G
z_1	g_1
z_3	g_2
z_5	g_3

d(Z,F)

Z	F
z_2	f_1
z_3	f_2
z_5	f_3

a(X,Y)

X	Y
x_1	y_1
x_3	y_2
x_3	y_3
~~x_6~~	~~y_3~~

b(X,A)

X	A
x_1	a_1
x_3	a_2
x_4	a_3

f(X,C)

X	C
x_1	c_1
x_3	c_2
x_5	c_3

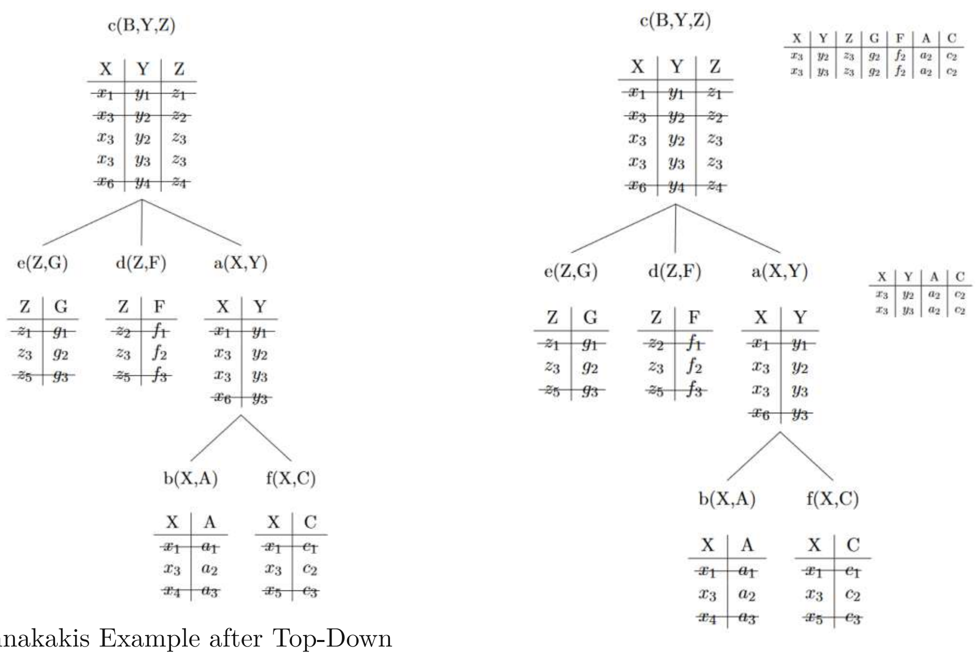
(b) Yannakakis Example after Bottom-Up Semi-Join

Figure 3.6: Yannakakis Example Part 1

along which the joins are, from the bottom up, performed. We put intermediate results of the joins next to the nodes for a and c in Figure 3.7b. The table next to c also represents a full enumeration of the queries result.

3.3.1 Detecting Empty Results

As discussed in Section 3.3, after step 1 of Yannakakis' algorithm, the bottom-up semi-join, we can determine if the query contains tuples in its result, by checking whether the result of the hypertree decompositions' root node is empty or not. In fact, this can be extended to nodes further down in the hypertree decomposition and their subtrees. Also, as the hypertree decomposition fulfills the connectedness condition, essentially giving the hypertree decomposition properties of a join tree, every hypertree decomposition nodes' data is backed by all of its children. Hence, if one of its children has an empty dataset, the parent will also have an empty dataset. Likewise, this phenomenon can be applied to all later stages of Yannakakis' algorithm. Bringing all pieces together, we can claim that, as soon as the (intermediate) result for one of the hypertree decompositions' nodes is empty, the final result of the complete query will also be empty.
This claim can be justified with a simple line of thought. As every semi-join or inner join, as we use them, must have two non-empty inputs in order to result in a non-empty output, we can conclude that in either case, as soon as some hypertree decomposition node's result is empty during the first step, the empty result propagates up to the root.



(a) Yannakakis Example after Top-Down Semi-Join

(b) Yannakakis Example Results Joined

Figure 3.7: Yannakakis Example Part 2

Furthermore, if the root's result is empty after the first step, we know that the query's result is empty. To show this, we can use the same line of thought for the second step, according to which an existing empty result of the first step propagates to the result of the second step. This also applies to the inner joins in the third step, finally giving us the empty result detected earlier.

3.3.2 Further Optimization

As mentioned earlier, after the first bottom-up semi-join pass, a query can potentially already be answered. This is the case for boolean conjunctive queries, as well as queries which can be answered through the schema of the base data, i.e., the relations in the edge cover λ, of the hypertree decomposition's root node. Ghionna et al. [GGGS07] experimented with this phenomenon. They created query-oriented hypertree decompositions as base for the query evaluation. These decompositions contain all relations needed for answering the query in its root node, in other words, all attributes are covered by the relations in the root's edge cover. They were therefore able to eliminate the second and third step of Yannakakis' algorithm during evaluation. A disadvantage is that the relations in the edge cover may not have an attribute overlap and thus cause a cartesian product, which could lead to an exponential blowup. We already mentioned this topic above, however, this time, it is more likely for the width to be higher. The reason therefore lies in the properties of the query-oriented hypertree decomposition, which is forced to cover all output attributes with its root. Additionally, Gottlob et al. [GLL+23] noticed that in such a case, set-safety has to be given for the result. Following, the multiplicity of the result tuples may not be fully guaranteed. Consequently, all distinct tuples are contained in the result, but tuples, which would occur multiple times when evaluating normally, due to multiple join partners in other hypertree decomposition nodes, are only contained once. Reconsidering the first step of the algorithm, we can see that semi-joins are performed, thus it is clear that the multiplicities in the result do not reflect the number of join partners.

3.4 Special Queries: 0MA

Additionally to the approach mentioned above, which tries to solve all queries with only one pass through the (query-oriented) hypertree decomposition, Gottlob et al. [GLL+23] identified a class of queries, which only requires executing the algorithm until the bottom-up semi-join, and therefore eliminates materialization of intermediate results. We already saw that this was sufficient for boolean conjunctive queries, and queries answerable through the hypertree decomposition's root node. Now, the 0MA queries extend the scope of applicable queries to queries including aggregates in the select list of a statement *select ... from ... where ... group by ...*, which is not the case in conjunctive queries.

To fully specify these queries, we have to introduce and define some terms. First, aggregation, respectively group by clauses, are denoted as $\gamma_U(...)$, where the subscript U defines the columns to aggregate and output, as well as aggregational operations. Aggregation normal form describes a query $Q = \gamma_U(\pi_S(Q'))$, where Q' is a query consisting of equi-joins and selections. As we already mentioned before, the root node of the respective hypertree decomposition of the query (with the relation R in its edge cover) has to contain all attributes which are included in the output, and additionally the group by clause. Hence, if there is such a relation R in the query, this relation R guards the query. Furthermore, set-safetyness has to be given, i.e., the result of a query $Q = \gamma_U(\pi_S(Q'))$ is equal no matter if the duplicates are eliminated before the aggregation, or not. Finally, we can define a zero-materialization answerable query as a query Q which is in aggregation normal form, guarded, and set-safe. [GLL+23]

With the conditions mentioned above in mind, we can directly infer that 0MA queries can contain the aggregation operations MIN and MAX, since these are set-safe. As an addition, COUNT can be combined with distinct in order to achieve set-safety. Consequently, an example 0MA query would be $Q = \gamma_{A,MIN(B),C}(a(A,B,C) \wedge b(C,D,E) \wedge c(D,E,F))$. The attributes to aggregate could be based in any relation, as long as the query is guarded by said relation. This is due to the fact that a hypertree decomposition of a 0MA query can be rerooted, without loosing any positive property. Indeed, rerooting is possible for any hypertree decomposition, independent of the type of query.

Apart from 0MA queries, other aggregates, like SUM, AVG, and COUNT without distinct could be implemented through counting algorithms, like Pichler and Skritek proposed in [PS13].

CHAPTER 4

Integrating Structure Guided Query Optimization into TiDB

In the previous chapters, we described how NewSQL databases work, as well as how queries can be optimized through structure guided methods. This chapter is going to describe the challenges faced during implementation of structure guided optimization into an existing database optimizer. Besides, we are going to describe the algorithms used for implementing this optimization.

We chose TiDB (`https://github.com/pingcap/tidb`), an Open-Source NewSQL database, to implement structure guided query optimization into. TiDB lies within the most popular Open-Source DBMS. Specifically, we are going to implement our approach into the database engine component of the TiDB ecosystem, which requires a substantial extension of the current database engine's query optimizer. As query optimizers are complex and highly efficient software, the biggest challenge of the work is to integrate structure guided query optimization as additional optimization alternative in the optimal places within the query optimizer, and reuse as much code as possible. Additionally, the existing functionality and performance of the database should not be disrupted through this integration.

On the way to implement the optimization, we had to investigate how TiDB, including query parsing, planning and execution, works. TiDB is quite a big project, therefore, getting hold of the mode of operations, and where the respective code lies in the code repository, is quite a challenge. Additionally, the documentation of the project is not the best, since we experienced a rapidly changing and growing development documentation, during our time working with TiDB, partially missing essential parts of the database's computational flow.

Furthermore, since structural query execution methods, especially Yannakakis' algorithm, slightly break with the traditional iterator executor model, as described before, we had to find a way to integrate the algorithm without adding too much overhead. This lies mainly in the repeated usage of intermediate results.

During the implementation and first tests, we collected several ideas on how to further optimize the approach implemented. These additional features and optimizations will be discussed in this chapter, including in which way they affect query planning and execution. Additionally, we are going to incorporate results from previous works attempting to integrate structural methods into databases, as well as theoretical inspirations, gained from literature research during the thesis.

We are going to publish our implementation on GitHub `https://github.com/MichaelMartinek/StructureGuidedTiDB`, and make it available as Open-Source software.

Summarized, we are going to document the means of operation, steps, and where this is hidden in the code repository, describe the added optimization for each step of the query execution pipeline, and briefly go over further optimizations, further improving our implementation.

4.1 Current Database Implementation

At the time we started implementing our optimized query execution approach, we took the current version of the main/master branch, which was *7cd2029* in the repository `https://github.com/pingcap/tidb`, corresponding to TiDB v6.7.0-alpha. We created a fork and inspected the means of operation of the database. The main language of the code base is Go. Fortunately, it is possible to run the database for testing purposes, without creating a full cluster and allocating the full resource set needed in a proper test or production database [Pin23e]. We outlined the big picture of the architecture in Section 2.3, including Figure 2.6 on page 17, which illustrates the different components of the database working together. For our optimization, we focus on the TiDB database computing component. With the mentioned possibility to run the database without all components, small and lightweight mocks are used to deliver the basic functionality of the database. With the same approach, the databases code can also be debugged, which is an important part of analyzing and understanding the current code, and eliminating any occurring bugs.

As a query optimizer, including the execution of a query, is handled by complex software, we have to get hold of the components working on these tasks, as well as where to start altering the software to achieve our goal. The development documentation of TiDB [Pin23e] gives a coarse overview of this pipeline, however, lacks in explaining some essential parts, as well as the depth needed to really understand the software. Therefore, it was important to go through the code manually, and also debug the code with certain inputs, to fully understand how the query execution works. The outcome of this analysis was a major step towards understanding how the query execution was realized, which will be explained below.

We already shortly discussed query planning and execution in TiDB in Section 2.3.3. In this section, we are going to discuss it more detailed. When speaking of a query, we mean an SQL DQL statement, querying data from the database.

A query passes three high-level layers when being executed. These layers are the protocol layer, managing the connection to the client, the SQL Layer, parsing, planning, and executing a statement from an SQL point of view, and the KV API Layer, which handles key-value requests originated from query plan operators/executors, forwards them to the right storage instance, and therefore realizing the distributed SQL features. Our focus will be the SQL layer, since it contains the query optimizer, thus, the most relevant part of the query execution workflow for our implementation. In Figure 4.1, the Protocol Layer and the KV API Layer are outlined, with the SQL Layer in between them, including further details. [Pin23f]

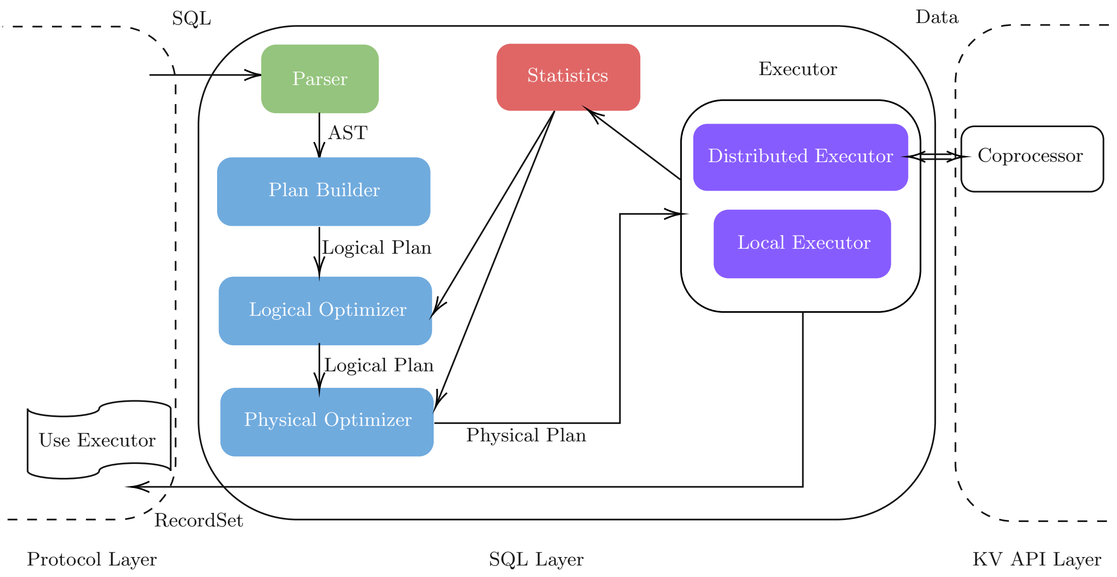


Figure 4.1: Current Implementation Architecture of TiDB - SQL Layer [Pin23j]

In general, a SQL statement, as string, is handed to the entry point of the illustrated workflow, shown by the arrow on the top left, denoted with SQL. At the end, a `RecordSet`, including the root executor, and providing methods to iterate the result, is returned. The corresponding interface definition is shown in Listing 4.1. This result is enumerated by the protocol layer, presenting the results to the user.

In the following pages, we will describe the means of operation of the SQL Layer, as shown in Figure 4.1. The green box in the illustration represents the parsing section, blue shows the planning part, and purple shows the executors. As an addition, database internal statistics, which are collected and queried, are illustrated by the red box.

The first step to handle an SQL statement is to parse it into a representation that can be performantly handled by the database. To this end, an AST is created. The

```go
type RecordSet interface {
        // Fields gets result fields.
        Fields() []*ast.ResultField

        // Next reads records into chunk.
        Next(ctx context.Context, req *chunk.Chunk) error

        // NewChunk create a chunk, if allocator is nil, the
↪   default one is used.
        NewChunk(chunk.Allocator) *chunk.Chunk

        // Close closes the underlying iterator, call Next
↪   after Close will
        // restart the iteration.
        Close() error
}
```

Listing 4.1: RecordSet Interface

respective parser realizing this functionality is located in the `parser` module of the code repository. A SQL statement is a concrete word, which conforms to the SQL language and grammar defined in the files `parser.y` and `hintparser.y`. The `hintparser` is mainly responsible for parsing optimizer hints [Pin23g], influencing how and what is optimized for a statement, as well as further properties about execution of a query, like for example the maximum execution time. Yacc, a parser generator, generates the respective `go` files realizing the parsing functionality. After parsing, the structure of the parsed statement is recreated using the types `ast.*`, such that all following steps handling the statement just access this data structure.

Next, the first planning step is conducted. Planning an SQL statement consists of the steps of building the initial plan, often named rewriting, logical optimization, and physical optimization. The module `planner` is the relevant module in the code repository for these steps. The `Optimize(...)` method in the file `optimize.go` acts as entry point for this process. For building the initial plan, originally, no optimized strategy, considering how the plan is built, is followed. The plan is built just as the statement was defined, without using any statistics, rules, etc. This is especially important for joins, which are unordered after this step. Initial logical plans are built using the `PlanBuilder`. Every plan is created by the `PlanBuilder.Build(...)` method in `core/planbuilder.go`. Methods for different constructs of statements, e.g., joins, selects, selections, aggregations, projections, etc., and their corresponding AST, creating logical plans, are defined in `logical_plan_builder.go` as members of `PlanBuilder`. We set our focus on the

PlanBuilder.buildSelect(...) method, which is the first method to be called for a select statement in the rewriting step, thus, being the optimal entry point for our implementation.

The resulting logical plan is then handed to the logical optimizer. The logical optimizer holds a list of rules, defined in core/optimizer.go, as optRuleList. Logical optimization generally is independent of the specific database instance and its data, applying tautologies to the current logical plan, without the use of database statistics and concrete data from the database. However, there are some instances of rules also using statistics, e.g., join reordering for cardinality estimation. Optimization rules implement an interface logicalOptRule, providing an optimize(...) method, in order to invoke the rules' specific optimization procedure. The procedures themselves are defined in multiple source files of format core/rule_*.go, including their rule names in the file name. The list of rules is enumerated, and each rule gets executed in the main method for logical optimization, logicalOptimize(...) in core/optimizer.go. The order of these rules is relevant, since they are executed in the same order as defined in optRuleList. Additionally, rules can be contained in the list, and therefore executed, arbitrarily often, which, for example, is needed for the column pruning, being added a second time at the last position.

Following, the logical plan is converted into a physical plan, that is chosen with knowledge of specifics of the database instance, including statistics. For each logical operator, the best physical operator is chosen, such that the cost of the whole plan is as low as possible. Physical operators differ in the way they access the data, whether it is reading an index, reading the whole table, or read with selection/functions as condition, as well as which type of storage is used. As we discussed in Section 2.3, TiDB supports different types of storage, with TiKV (row-based), and TiFlash (column-based), as options. Our primary focus for the implementation of structure guided query evaluation will be on row-based storage. Furthermore, they differ in their (or their respective executors') concrete implementations, which are used for different prerequisites, and obviously have differing costs. One example would be different join implementations, like HashJoin and MergeJoin. A MergeJoin requires the data to be ordered, whereas a HashJoin does not. In addition, they will have different costs, due to different computational requirements. The goal of physical optimization is to get a plan realizing the given logical plan, with the lowest cost possible. Again, core/optimizer.go contains the starting point for physical optimization, physicalOptimize(...). Inside this method, LogicalPlan.findBestTask(...) is called for the root executor. findBestTask(...) must be implemented by every logical plan operator, some of which implicitly implement it, some use the default implementation of baseLogicalPlan. Some of the logical plans implicitly implementing the method include a DataSource, LogicalMemTables, LogicalDualTables, and LogicalCTE(-Tables). Logical plans for which the LogicalPlan.findBestTask(...) method is implicitly defined either only have one possibility, and therefore nothing to optimize in this stage, like DualTables, MemTables, and Common Table Expressions (CTEs), or require special handling, like DataSources. For `baseLogicalPlan`, physical plans are created for one logical plan node in `LogicalPlan.exhaustPhysicalPlans(...)`, which are recursively built in `baseLogicalPlan.enumeratePhysicalPlans4Task()`, calling `findBestTask(...)` for the children of this plan. Therefore, the best plan, according to its cost is found.

As soon as the optimized physical plan is found, it gets converted into executors, implementing the actual procedures for executing the physical plan nodes. Executors work based on the iterator concept [Gra94], as we mentioned earlier. The interface `Executor`, as shown in Listing 4.2, defines the methods realizing the iterator functionality for every executor, since all executors must implement the interface in order to be usable. Executors are built, just like (physical and logical) plans, in a tree-based structure, taking results from their children and hand new results to their parents. Chunks, a set of rows of data, are used as unit of iteration. The idea behind the iterator concept is to consume every chunk just in the moment of creation by the child executor, eliminating the need to store this data in memory or storage. However, some executors need to store data in order to operate, these are for example a HashJoin, storing the HashTable and associated rows, or a CTE. Every executor is free to choose how to realize their functionality, whether they compute their result single-threaded, or multi-threaded, respectively, in multiple parallel go routines, storing results, or do a just in time computation without storing the results. All executors are located in the `executor` module in the code repository.

```go
type Executor interface {
        base() *baseExecutor
        Open(context.Context) error
        Next(ctx context.Context, req *chunk.Chunk) error
        Close() error
        Schema() *expression.Schema
}
```

Listing 4.2: Executor Interface

To handle some universal scenarios and collect statistics, `executor.go` provides a `Next(..., e Executor, ...)` wrapper function, calling the executor's `e.Next(...)` method after finishing these tasks.

Executors can be classified into two types, local executors, which run directly and entirely on the TiDB database node, and DistSQL (distributed SQL) executors, including TableReaders, IndexReaders, etc. Distributed SQL executors access the distributed storage provided by the storage engines TiKV (row-based) and TiFlash (column-based). Additionally, a coprocessor is provided by the storage, enabling the distributed and parallelized execution of tasks on these nodes. With the coprocessor, certain tasks, like fetching rows only if an expression, including selections with simple scalar values, other

columns (in the same or other tables), or functions, applies, or computing an aggregate, can be executed directly on the storage nodes. It enables the simple distributed storage to be a distributed storage and computing component.
DistSQL executors are contained in `distsql.go`, among other source files. Since we focus on the database node itself, they are of minor importance for our implementation. Other executors are scattered across the module, some of which include the HashJoin executors, in `join.go`, CTE and CTETable executors in `cte.go` and `cte_table_reader.go`, and HashAggregates in `aggregate.go`.

To complete the circle for the description of the database, the root executor, as mentioned before, is then returned in the `RecordSet`, to iterate the result.

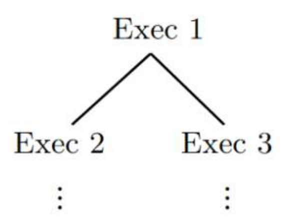
Optimization as it is in TiDB is deterministic, leading to the same plans after the optimization process without change of the actual data, parameters, or database instance. Following, execution times for queries are stable, with possible improvements on repeated executions due to cached results. Further properties influencing the optimization can be set through optimizer hints. Some operators, like specific join implementations, could be excluded or forced to be used. Depending on the effects, hints are used during initial plan creation, logical optimization, or physical optimization, i.e., on the whole optimization process.

4.2 Integrating Structure Guided Query Optimization

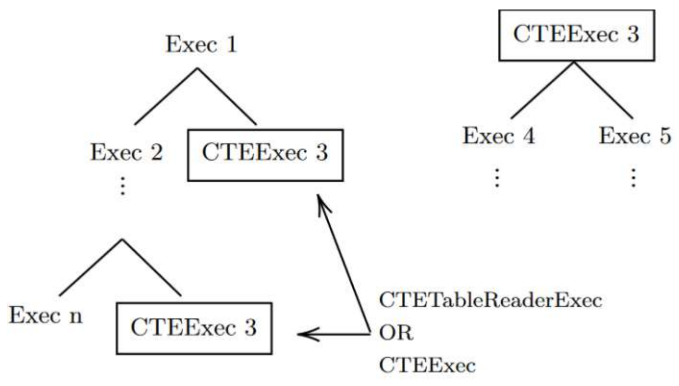
As we discussed the process of structure guided query optimization in Chapter 3, we will now integrate actual structure guided query optimization into TiDB. Other than in the theoretic position we took in Chapter 3, we now integrate the approach into an existing DBMS. Therefore, we cannot abolish the stages of logical and phyiscal optimization including some join reordering, etc. We will instead integrate structure guided query optimization and evaluation into the existing workflow.

One component we will take as given is the creation of hypertree decompositions from query hypergraphs. This will be solved by the software BalancedGo, published as Open-Source software `https://github.com/cem-okulmus/BalancedGo`, being a result of research by Gottlob et al. [GOP22, GLOP22]. All other steps will be realized directly in TiDB. A list of these steps are:

1. Parse the query to a query hypergraph
2. Decompose the query hypergraph with BalancedGo into a hypertree decomposition
3. Create the query plan realizing the procedure of Yannakakis' algorithm, as shown in Section 3.3
4. Execute the created query plan



(a) Execution Tree without CTEs



(b) Execution Tree with CTE: CTE Defines a new Tree which can be Used in Main Tree

Figure 4.2: CTE Usage Example

We already saw in Section 3.3, that some of the intermediate results need to be accessed multiple times during execution of Yannakakis' algorithm. This contrasts with the iterator concept we discussed earlier. Therefore, we had to either find an existing plan node enabling repeated usage of subplans, or create a new one. We chose CTEs as the best concept to satisfy our requirements.

CTEs can be seen as stored intermediate result in a plan. They change a simple plan, in tree structure, to a forest of trees, one of which is the main tree, representing the root executor. Results of other plans/trees can be read multiple times. Figure 4.2 illustrates the difference between an execution tree with (Figure 4.2b) and without CTEs (Figure 4.2a).
The respective components in the code, in module `executor`, are `CTEExec` in `cte.go`, representing the root of a tree, and `CTETableReaderExec` in `cte_table_reader.go`, capable of reading a previously created CTE. Every CTE has an ID, which identifies the CTE and its storage. E.g., a `CTEExec` has ID 1, then the `CTETableReaderExec` with ID 1 reads the same storage. Interestingly, `CTEExec` acts as a combined writer and reader, such that at the first call of `Next(...)` for a `CTEExec`, as long as no other `CTEExec` with the same ID had calls to `Next(...)`, the seed executor, the underlying executor, is entirely executed and the result stored in memory. Afterwards, the first chunk is returned. On every subsequent call to the same `CTEExec`, the result is further iterated, and chunks are returned. Other `CTEExec` executors referring to the same storage ID do not execute the seed executor, but rather access the storage and iterate and return the result chunks, just as a `CTETableReaderExec` would do. CTEs have the disadvantage, that they can be very memory-intensive because of the intermediate result being stored in memory. However, we need to use the results several times to replicate Yannakakis' algorithm. Hence, we are willing to accept this drawback.

4.2.1 Basic Implementation

The core of our implementation is a query rewriting mechanism, manipulating the initial plan built. Figure 4.3 illustrates the main steps and decisions for the plan creation. Orange squares are branching components, blue nodes are operations. Furthermore, the illustration shows how the cases to use structure guided query evaluation, or not, and 0MA queries, are handled.

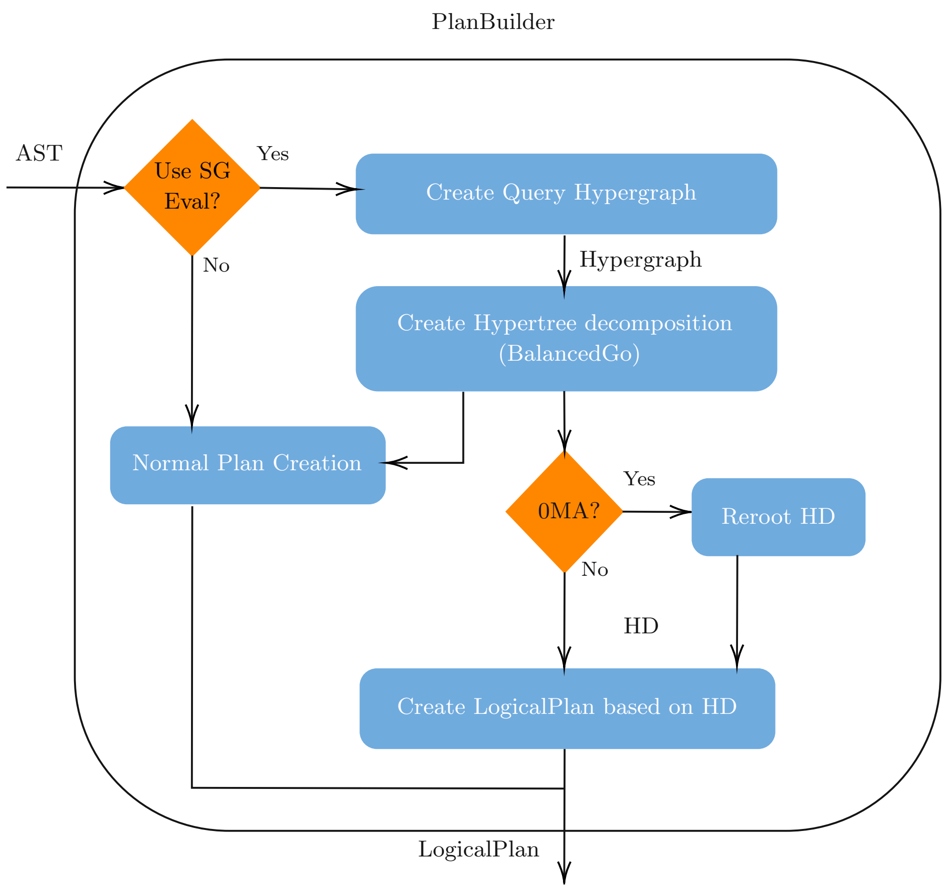


Figure 4.3: Plan Builder with Structure Guided Query Evaluation

Initially, the plan builder receives an AST of the query. Based on the AST, a check whether the given statement should be solved with structure guided query evaluation, is conducted. This decision is done based on the components of the query. Additionally, we already check if the statement is a 0MA query, and prepare several properties needed for 0MA query computation, which will be used further into the process.

We already inspected the `logical_plan_builder.go` to find the optimal entry point for handling structure guided queries. Our implementation starts in `PlanBuilder .buildSelect(...)`, since all `select` statements pass through this method.

The first step, creating the query hypergraph from the AST, is realized similar to an

Open-Source SQL to Hypergraph converter written in Java `https://github.com/dmlongo/sql2hg`. For performance reasons, we created a new native go implementation including `buildHypergraphForSelect(...)` in `logical_plan_builder.go` and the module `util.hypergraphBuilder`. As shown in Listing 4.3, first, every table of the statement is registered in the hypergraph builder, followed by all equality conditions. The hypergraph builder uses a UnionFind data structure to compute overlapping parts of the hypergraph. Finally, a string representation of the hypergraph, as well as the mapping of variables (which are used in the hypergraph instead of columns) to column names, are created and returned.

```
tables = getTables(ast)
eq = getEqualityConditions(ast)
hgb = initHypergraphBuilder(getUniqueExpressions(eq))

for t in tables:
    hgb.buildEdgeInit(t)

for c in eq:
    hgb.buildEdge(c.left.table, c.left.column)
    hgb.buildEdge(c.right.table, c.right.column)
    hgb.buildJoin(c)

return hgb.MakeHypergraph(), hgb.GetMapping()
```

Listing 4.3: Pseudocode for Creating a Hypergraph

Before creating the hypertree decomposition, all hypergraph edges without vertices, e.g. `relA()`, are removed. For these relations, a cartesian product has to be built. Calculating the cartesian product will be conducted after the last join of the plan, to minimize the number of rows in intermediate results.

Next, the created hypergraph has to be decomposed into a hypertree decomposition. Therefore, we use the software BalancedGo `https://github.com/cem-okulmus/BalancedGo`, which can either be used as standalone executable, or as go library, since BalancedGo itself is written in go. With BalancedGo, we use the `BalSepLocal` algorithm to create a hypertree decomposition. The decomposition and algorithm itself are taken as given, without further looking into specifics of the used algorithms.

Two cases which we want to avoid, are empty or duplicated hypertree decomposition node covers, or empty elements in node bags. Hypertree decompositions with one of these two properties will not be accepted. Thus, they are recomputed until an acceptable decomposition is found. The term acceptable also includes the correctness of the hypertree decomposition. For performance reasons, we limit the number of tries to 10. BalancedGo uses some randomness in its computations, making repeated tries a necessity, since every hypertree decomposition can potentially be different. Considering that the hypertree width for the intended decomposition has to be set manually for BalancedGo, and that the lower the width, the better the performance of the respective plan, we first attempt to find decompositions for hypertree width 1. If there is no acceptable decomposition, the same procedure is carried out for hypertree width 2. In case neither of the two options yields an acceptable decomposition, the optimizer reverts to normal query planning.

Now, as we mentioned in Section 3.4, 0MA queries require the hypertree decomposition to be rooted at a particular node. This node must contain the guard relation in its edge cover, such that one traversal with Yannakakis' algorithm suffices to get all result tuples. Consequently, the hypertree decomposition has to be rerooted for a 0MA query in case the guard is not contained in the root node.

Based on the hypertree decomposition, a logical plan is built. For this reason, we have defined three traversal algorithms, that represent the three steps of Yannakakis' algorithm, i.e., bottom-up semi-join, top-down semi-join, bottom-up join, traversing the hypertree decomposition. These traversal algorithms build the logical plan step by step. For 0MA queries, the first bottom-up traversal is sufficient.

The first bottom-up traversal is defined in Listing 4.4. `DecompRoot` is the current root node of the hypertree decomposition, `isRootNode` is a Boolean parameter, set to true at the first call, indicating that the current node is the root node. Note that `buildSemiJoin(planA, planB, expr)` always builds a semi-join `planA` $\ltimes$ `planB`, whose result schema is identical with `planA`.

First, we check for a special case in which the root node of the hypertree decomposition contains all relations in its cover, without children. In this case, the relations in the cover are joined via semi-joins and the result is saved as CTE for the respective relation. The schemes of the CTEs and the returned plan are identical to the schema of the relation dealt with last.
Without this special case, a logical plan node for the current hypertree decomposition node's DataSource is created. If the node contains more than one entry in its cover, these relations are joined first by `getCTEOrDatasource(decompRoot)`. Selections applicable to single relations are added at creation of the DataSource, to keep the number of tuples in intermediate results as low as possible. For each child, the method is recursively called, followed by building a semi-join `decompRoot` $\ltimes$ `child`, such that the resulting schema only contains `decompRoot`'s columns, representing an upwards semi-join in the hypertree decomposition. The recursion itself also follows a bottom to top order. Before returning the created logical plan, a LogicalCTE operator is added to the plan for internal nodes of the hypertree decomposition, such that these results can be reused in later steps. The logical plan returned in this traversal represents the bottom-up semi-join pass of Yannakakis' algorithm, which is sufficient for 0MA queries, as well as existential queries. For these types of queries, leftover relations without join conditions, i.e., causing cartesian products, are added, as well as aggregates and projections to complete the respective computation.

```
buildPhase1(decompRoot, isRootNode):

    // special case
    if isRootNode and len(decompRoot.Cover) > 1 and
     ↪  len(decompRoot.Children) == 0:
        for rel in decompRoot.Cover:
            if currPlan == null:
                currPlan = buildDataSource(rel)
            else:
                newPlan = buildDataSource(rel)
                expr = buildExpressionsForJoin(decompRoot,
                 ↪  decompRoot)
                currPlan = buildSemiJoin(newPlan, currPlan,
                 ↪  expr)
            currPlan = buildCTE(currPlan, rel)
        return currPlan

    currPlan = getCTEOrDatasource(decompRoot)

    for each child of decompRoot:
        childPlan = buildPhase1(child, false)
        expr = buildExpressionsForJoin(decompRoot, child)
        currPlan = buildSemiJoin(currPlan, childPlan, expr)

    if decompRoot is not a leaf:
        currPlan = buildCTE(currPlan, decompRoot)
    return currPlan
```

Listing 4.4: Pseudocode for First Bottom-Up Traversal of Plan Builder

As an example, we are going to define a hypertree decomposition in Figure 4.4. We are going to assume that the hypertree width is 1, and name the relations according to the numbering of the hypertree decompositions nodes. Moreover, bags are omitted, since we are just going to show the structure of the created join trees, not the conditions. The intermediate plans after finishing the first phase are illustrated in Figure 4.5. Note that the plans are created from left to right, and as soon as a plan for a relation is created, it is used as CTE in all following plans (illustrated with an underline). Empty nodes represent intermediate results of joins or semi-joins. In this stage, we are only conducting semi-joins, in which, the left child defines the schema of the intermediate result after the join. Also, the plan for 1, illustrated in Figure 4.5b will be directly returned by `buildPhase1(decompRoot, isRootNode)`.

The next traversal, the top-down step, is defined in Listing 4.5 as pseudocode. Here,

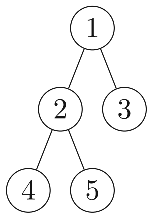
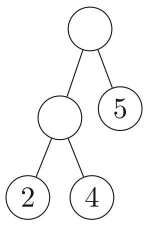
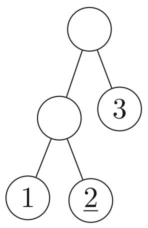


Figure 4.4: Example of a Simplified Hypertree Decomposition



(a) Join Tree for 2



(b) Join Tree for 1

Figure 4.5: Resulting Join Trees After Bottom-Up Semi-Join

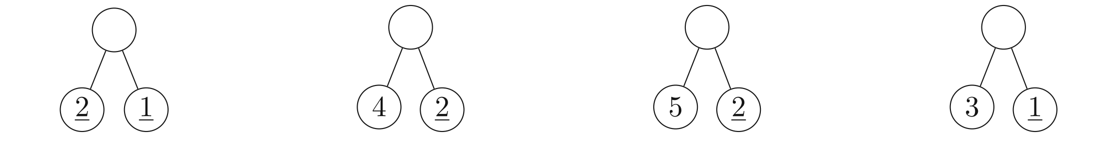
again, we have to consider the same special case as in the first traversal, in which the root node of the hypertree decomposition contains all relations in its cover, without children. Since the first traversal iterated over the hypertree decomposition node's cover from the beginning to the end, and used the respective last semi-joined relation as schema, `plan` initially contains this schema. Therefore, we can skip this relation during the top-down traversal. Opposite to the first traversal, the relations are semi-joined in reverse order with the previously created CTEs. These result logical plans are stored as CTEs, with the last one being directly returned.
Otherwise, we conduct semi-joins with each child of the current hypertree decomposition node, such that the child defines the schema, i.e., a downward semi-join in the hypertree decomposition is carried out. For the semi-join with the leftmost child, we can directly use the given `plan`, representing the query plan for the current hypertree decomposition node `decompRoot`. For all semi-joins with subsequent children of the current node, the same data has to be reused. Therefore, a CTE(-Reader), with which the data can be read again, is created. After building the semi-join, the result thereof is saved with a CTE, to be made available for the following steps. The just saved plan, is then used to recursively call the method for the current child, creating a `subPlan`, which is either set as `leftMostPlan`, in case the current child is the leftmost child in the leftmost branch of the hypertree decomposition, or saved as plan for the current child. The recursion defines the top to bottom traversal. Finally, the `leftMostPlan` is returned. Considering a tree, and calling the method for the root of the tree as `decompRoot`, the final result will be the plan for the leftmost and deepest node. Plans saved as `subPlans` are used in the third phase.

Continuing our example, the plans created during the second phase of the algorithm for our simplified hypertree decomposition in Figure 4.4, are illustrated in Figure 4.6. As in the previous phase, all illustrated joins are semi-joins. Note that the only new plans, i.e., using DataSources, visible by not being underlined in the figure, are the leaf nodes of the hypertree decomposition. These are not included in the bottom-up semi-join phase. For this reason, CTEs are saved for them for the first time during this stage. The plan for node 4 will be returned as the `leftMostPlan` at the end of the stage. All others are made available via CTEs.

```
buildPhase2(decompRoot, plan, isLeftMostBranch, isRootNode):

    // special case
    if isRootNode and len(decompRoot.Cover) > 1 and
     ↪  len(decompRoot.Children) == 0:
        for rel in decompRoot.Cover, penultimate to first:
            newPlan = getSubplanOrCTE(rel)
            expr = buildExpressionsForJoin(decompRoot,
             ↪  decompRoot)
            plan = buildSemiJoin(newPlan, plan, expr)
            plan = buildCTE(plan, rel)
        return plan

    leftMostPlan = plan
    for each child of decompRoot:
        if child is not the leftmost child:
            plan = getCTE(decompRoot)
        childPlan = getCTEOrDatasource(child)
        expr = buildExpressionsForJoin(decompRoot, child)
        newPlan = buildSemiJoin(childPlan, plan, expr)
        newPlan = buildCTE(newPlan, child)

        subPlan = buildPhase2(child, newPlan, child is leftmost
         ↪  child and isLeftMostBranch, false)

        if child is leftmost child and isLeftMostBranch:
            leftMostPlan = subPlan
        else:
            save subPlan for node child
    return leftMostPlan
```

Listing 4.5: Pseudocode for Top-Down Traversal of Plan Builder

The final traversal step for building the plan is defined in Listing 4.6. This step unifies the previously prepared subPlans and CTEs to a complete plan. Just as the first



(a) Join Tree for 2 (b) Join Tree for 4 (c) Join Tree for 5 (d) Join Tree for 3

Figure 4.6: Resulting Join Trees After Top-Down Semi-Join

bottom-up traversal, this traversal again passes the hypertree decomposition from the bottom to the top, joining along the structure of the hypertree decomposition. Also, the special case in which the root node of the hypertree decomposition contains all relations in its cover, without children, has to be handled again. For this reason, we traverse `decompRoot.Cover`, get a subplan or CTE for each relation, and join them together with inner joins. The final join tree is then returned. Just as in the top-down traversal, we skip the first element of the cover, since it is already included in the given `plan` parameter.
In any other case, we traverse the hypertree decomposition such that we reach the bottom, i.e., the current `plan`'s CTE name, which was the result of the second step, matches with the decomposition node. This plan is then returned and used in the parent. For all other children, subPlans are used or CTE(-Readers) (to use the data) have to be created. The children are one after each other inner joined with the parent, until the root is reached again, and the join plan is finished.

Also, finishing our example, the final plan created for our simplified hypertree decomposition in Figure 4.4, is illustrated in Figure 4.7. Note that at this point we are only conducting inner joins. Additionally, as we can see, all the nodes are underlined now, meaning we use the plans created in the previous phases.

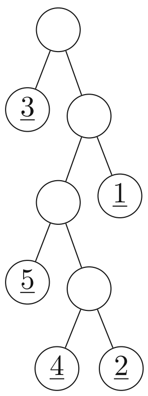


Figure 4.7: Resulting Join Trees After Bottom-Up Join

One essential part we abstracted until now is the method `buildExpressionsForJoin (hdNode1, hdNode2)`, creating expressions for joins between these two hypertree decomposition nodes. Join expressions are determined by a certain procedure. For this

```
buildPhase3(decompRoot, plan, isRootNode):

    // special case
    if isRootNode and len(decompRoot.Cover) > 1 and
     ↪  len(decompRoot.Children) == 0:
        for rel in decompRoot.Cover, skip first element:
            newPlan = getSubplanOrCTE(rel)
            expr = buildExpressionsForJoin(decompRoot,
             ↪  decompRoot)
            newPlan = buildInnerJoin(plan, newPlan, expr)
        return newPlan

    if plan is a CTE for decompRoot:
        return plan

    currPlan = getSubplanOrCTE(decompRoot)

    for each child of decompRoot:
        childPlan = buildPhase3(child, plan, false)
        expr = buildExpressionsForJoin(decompRoot, child)
        currPlan = buildInnerJoin(childPlan, currPlan, expr)

    return currPlan
```

Listing 4.6: Pseudocode for Bottom-Up Join Traversal of Plan Builder

purpose, the bags of the two involved hypertree decomposition nodes are intersected. We then collect the involved columns for each element of the intersection in different lists. Each of the lists defines columns which should match. Consequently, each list is traversed for elements occurring in the schemes of the given plans. If two elements occur as columns, an equality expression is created for these columns.
Additionally, for cyclic queries, in the third phase of Yannakakis' algorithm, we have to check for columns which originate from the same relation, and occur in some variable mapping (mapping bag variables to columns), or are primary keys in their original relation, and create equality conditions for them, to avoid duplicated result tuples. This is necessary, because for this kind of queries, it is possible that data originating from the same relation is joined with each other. Due to aliases created for CTEs, columns originating from the same original table and original column could be contained multiple times across the schemes of the left and right plan. To avoid duplicates resulting from this phenomenon, and therefore wrong query results, the respective columns have to be compared with each other. Most importantly, legitimate duplicates originating from bag semantics are not affected by this measure. Finally, the complete list of equality

conditions is returned to be used for (semi-)joins.

We added an example for the above explained procedure in Figure 4.8. The illustration shows two hypertree decomposition nodes, with their bags' contents. To understand the example, we just need the mapping for v5, which contains relA.ID, relB.FK and relC.FK. The intersection of vertices just contains v5. For this reason, all elements occurring in the respective schema, i.e., relA and relB, are taken to build equality conditions, with the result as shown in the illustration.

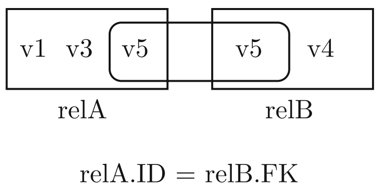


Figure 4.8: Example for Building an Expression Between Two HD Nodes

Finishing the created query plan, just as for 0MA queries, we still need to add cartesian joins, if there are any, and a projection to project any duplicate columns resulting from CTE aliasing, or unused columns, like artificial row IDs, away. This plan is then handed to the further steps of the optimizer, as illustrated in Figure 4.1. The optimizing procedure may include some form of join reordering, optimizing to equivalent query plans, e.g., by changing the inner join order of the third phase of Yannakakis' algorithm, predicate pushdowns, and elimination of redundant expressions, as well as physical optimization.

The plan created by these algorithms already implements the essentials of structure guided query optimization and evaluation, including Yannakakis' algorithm. However, we added other features, making controlling the execution easier, and further optimized query evaluation, using database internals.

4.2.2 Further Features and Optimizations

During planning of our implementation and testing, we realized there are multiple possibilities to give the user more control over the usage of structure guided query optimization and execution, and further leverage internal database mechanisms, as well as generate hypertree decompositions with higher quality. We define the quality of a hypertree decomposition as the ability of the implied query plan to quickly reduce the number of redundant tuples. Hence, we implemented them and tested these possibilities for their usefulness. Our additionally implemented features and optimizations include the following ones:

- Optimizer hint to disable structure guided query optimization and evaluation
- Optimizations in handling CTEs
- Optimizations in case it is foreseeable that the query result will be empty

- Optimizations regarding the stability and quality of the hypertree decompositions and the associated query plan

Optimizer Hint to disable new Implementation

The first feature is an optimizer hint, giving the user more control over the used optimization and execution options. As other optimization features and executors can be enabled and disabled via hints, we added a feature to disable our implementation. The respective optimizer hint `YAN_NORMAL()` can be used just as any other optimizer hint, after the `SELECT` keyword: `SELECT /*+ YAN_NORMAL() */ * FROM ....` In Section 4.1, we already discussed how to manipulate the query parsing process in order to add new optimizer hints.

This feature is not only useful for the actual end users, but also for our testing process, as it enables us to switch between normal execution and structure guided execution without patching and restarting the database.

CTE handling Optimization: CTEMock

Considering we are heavily using CTEs for our plans, and some of these CTEs are only used once, we developed a new CTE executor, optimizing execution for these cases. CTEs work in a way, such that at the first call of `Next(...)` for the `CTEExec`, the whole seed plan is executed, and the result is stored in memory. This is, depending on the amount of data, very resource intensive, especially if some of these CTEs are only used once. So, instead of using `CTEExec` for these CTEs, we developed a new executor, `CTEMockExec`, which just hands `Next` calls to the seed plan, executing the seed plan just in time, instead of in advance, and passing-through the chunks. As we noticed during testing, the memory resources were reduced through this measure. Although there is an existing capability of the database to 'merge' CTEs into plans, this did not work properly for us, and we had to implement our own solution to this problem.

To check which CTEs are only used once, we traverse the logical plan after creating it, and count the occurrence of every CTE ID. The counts are saved in the statement context, from which they can be retrieved at any step during planning and execution. When building the executor from the physical plan, in `executorBuilder` in `executor/builder.go`, specifically the executor matching the `PhysicalCTE` plan, we check these counts. In case the count for the current ID is 1, we return a `CTEMockExec` instead of an `CTEExec` in `executorBuilder.buildCTE(...)`. During execution, the `CTEMockExec` is then used for computing the respective CTE.

Early Stop

During testing, we observed long runtimes on queries with an empty result, compared to normal execution. Given that our test queries and dataset cover this case pretty well, we had strong evidence of a problem to fix.

We already discussed how to detect empty results early during query evaluation through Yannakakis' algorithm in Section 3.3.1. The same technique is now integrated into our implementation to try to solve the aforementioned problem.

In our implementation, we can use the CTEs to check for empty intermediate results. We implemented the detection for `CTEExec`, `CTEMockExec` (and additionally for `IndexLookUpExecutor`), in whose `Next(...)` methods, we set a flag in the statement context as soon as an empty intermediate result is found. Conveniently, the flag can be checked for all executors in the `Next(...)` wrapper function, that we discussed in Section 4.1 on page 46. Once the flag is set, only empty chunks are returned by all executors. This is especially important for executors in other branches of the query plan. As we cannot revoke previously returned chunks for other executors, this measure primarily reduces the amount of data to be worked on. Eventually, however, the empty result, which would also appear without setting the flag sooner or later, will propagate to other executors. With this measure, we aim to reduce the runtime for the aforementioned queries.

Due to the fact that an empty result can be detected at any point during query computation, it is possible that indices and the following readers reading the respective tuples return a differing number of tuples. Therefore, consistency checks, e.g., comparing the number of returned tuples of IndexReaders, and their following TableReaders reading ranges returned by the IndexReader, need to be incorporated into this feature. Since the earlyStop feature potentially changes the number of returned tuples, we had to add an exception to these checks.

Quality of the Hypertree Decomposition and Query Plans

As it turns out, the quality of the hypertree decompositions potentially differs in each run, with the degree of difference dependent on the size of the query, and the width of the hypertree decomposition. As we mentioned at the end of Section 4.1, normally, execution times are consistent. Not so with structure guided query optimization and execution, as we originally realized it. In our initial implementation, we used BalancedGo's `BalSepLocal` algorithm, without the possibility to add inputs considering how expensive joins between different relations are. However, the fact that BalancedGo uses algorithms tailored towards a lower resource usage and finding an acceptable hypertree decomposition in reasonable time, instead of finding the best one, as we discussed in Section 3.2.2 on page 33, makes it necessary to add further data to the computation. Consequently, we switched the algorithm to `JCostBalSepLocal`, allowing us to take join costs into account. This approach was already used by Gottlob et al. in [GLL$^+$23]. Whereas these join costs help with the general quality of the hypertree decompositions, they do not rule out the occurrence of bad ones, rather just lowering the probability of bad hypertree decompositions being created. One problem that was nearly completely solved through this step was the occurrence of cartesian products in the plans of cyclic queries. Cartesian products occur in case the hypertree width is 2, and the respective relations in the hypertree decomposition nodes' cover are not sufficiently covered by the vertices in the bag, such that there exists no join condition between them. One essential part responsible for this was how we created the join costs. The costs are created based on the hypergraph of the query. Each pair of relations and their vertices are compared. Costs for a potential join between each of the involved relations are defined as follows:

- In case there are no conditions applying to a potential join, we set the cost to 100000000.0, since this reflects a cartesian product, which should be avoided.
- If there is a direct relationship, meaning the vertices applying to one relation are a subset of the vertices applying to the other relation, we set the cost to 1.0.
- For any other relationship, the number of rows in the table is taken into account: $1 + (1 + \frac{\text{rowCount1}}{10000}) * (1 + \frac{\text{rowCount2}}{10000})$. The goal of the formula is for the cost to be above the 1.0 of direct relationships (including some difference), below 100000000.0 of cartesian products (hence, we use $\frac{rowCount}{10000}$), and to reflect the worst case result rowCount, by multiplying the rowCounts of the involved relations, with guarantee that the factors are at least 1.0, to create a difference compared to a direct relationship. In any case, the resulting cost is at least 2.0.

Row counts are retrieved from the databases internal statistics, slightly changing the architecture of the plan builder, as shown in Figure 4.9.

Although the above-mentioned optimization improves the quality of hypertree decompositions, their quality still fluctuates from run to run. To improve this situation, and even out these fluctuations, we added another feature, secondaryPlans. SecondaryPlans creates three plans instead of one during plan building. After physical optimization, the plan's costs are compared and the best one is chosen for execution. We classify this feature as part of the physical optimizer, since this is when the final decision which plan to use is made. The main manipulations for computing secondary plans are contained in `PlanBuilder.buildSelect(...)`, building the initial secondary plans, as well as `optimize(...)` in `planner/optimize.go`, where the creation of the physical plan is invoked, and the decision for a plan is made.

Originally, we planned to fully parallelize the creation of secondary plans, their hypertree decompositions, and the respective optimizer stages. However, it turned out that the optimization process heavily relies on a set of shared context variables, which did not allow us to parallelize the optimization process. Additionally, BalancedGo, as we use it, does not seem to be fully designed to be used for inter-query parallelization, which not only includes decomposing multiple queries at once, but also creating multiple decompositions for the same query at once. This is due to the usage of global variables, especially in the parsing process, through which different go routines influenced each other. This problem is, as already mentioned, not restricted to secondary plans, but also existent for concurrent operation of the database, working on multiple queries at the same time. A solution for this problem would be to use a precompiled binary of BalancedGo, and invoke it in child processes. Still, we decided to use BalancedGo as library for now, as our testing procedure does not include concurrent queries.

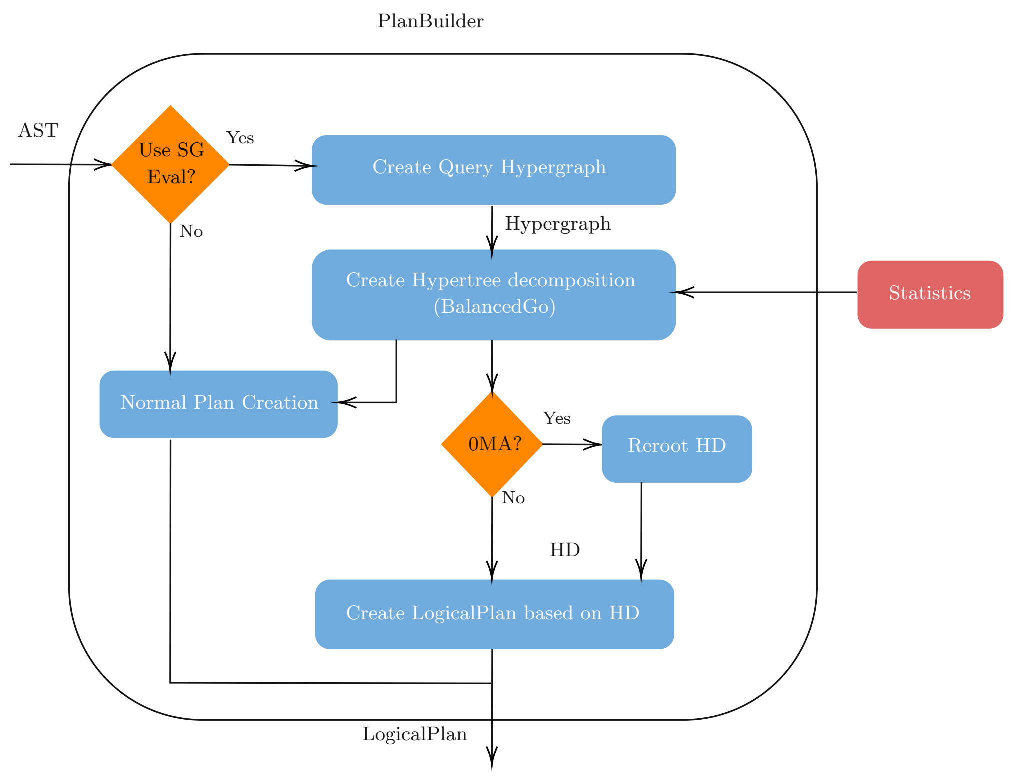


Figure 4.9: Plan Builder with Structure Guided Query Evaluation Including JoinCosts

4.3 Final Architecture

At the end, we want to take a look at the final architecture, and which components in the flow of processing the query were changed. In Figure 4.10, we marked the changed components with a red highlight on the illustration. An optimizer hint was added to the parser, followed by the main query rewriting in the plan builder, decomposing the query and realizing Yannakakis' algorithm in the form of a query plan. The decomposition with join costs also incorporates database statistics. Secondary plans are created in the plan builder. Subsequently, the final plan is chosen in the physical optimizer. Lastly, local executors are modified to include the `CTEMockExec`, easing resource usage during query execution. In addition, for queries for which it is foreseeable that the result will be empty, the resource usage is reduced with the earlyStop feature.

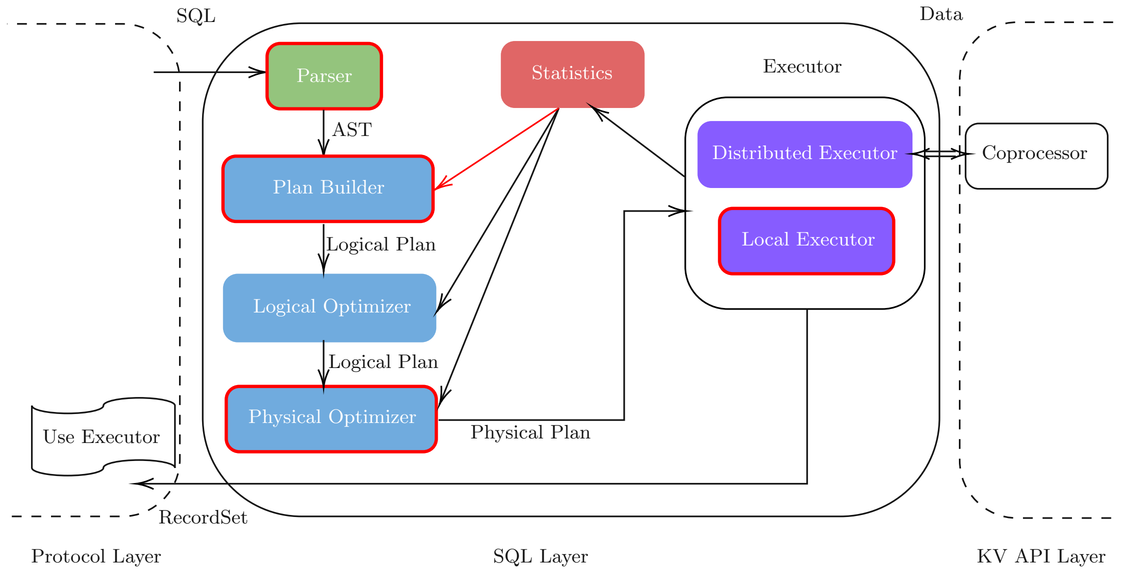


Figure 4.10: Implementation Architecture of TiDB with Changes marked Red - SQL Layer [Pin23j]

CHAPTER 5

Experimental Evaluation

In this chapter, we are going to report on experimental evaluations on several aspects of our implementation, beginning on the usefulness of our additional features, assessing their performance and benefits compared to not having these features. Moreover, we will evaluate the complete implementation including all beneficial features. We are going to determine reasons for the results, and conduct a discussion about them.

The goal of this chapter is to collect data of the performance of our implementation, compared to normal query execution, enabling us to draw a conclusion whether structure guided query optimization and execution, as we integrated it in TiDB, provides benefits for query execution.

5.1 Methodology

The methodology of our tests and evaluations mainly includes the setup in terms of hardware and software, the data we use for our tests, and the associated queries. In some scenarios, the used software versions, especially the used TiDB database engine, and the execution modes, are different to others. Additionally, depending on the goal of the test, all queries or subsets thereof are executed. However, these cases will be marked and explained in their respective sections.

5.1.1 Setup

Our setup consists of a cluster of VMs with specifications as listed in Table 5.1. Our setup is oriented on the recommendations by PingCap for a TiDB test cluster [Pin23i]. Moreover, we modified the amount of resources of some machines after conducting initial tests. The machines of the cluster run Ubuntu 22.04 LTS. Our implementation is compiled on one of our machines in the test cluster, Monitor01, running Go version 1.20.6. We use TiUP to manage the TiDB database cluster. The base version of the TiDB cluster

Host	CPUs	RAM	Storage	Roles
Monitor01	4	8 GB	Basic 100 GB Disk for Repo 200 GB Disk for TiDB	TiUP Test-Manager & Executor Build new TiDB patches
Monitor02	4	8 GB	Basic 200 GB Disk for TiDB	TiDB Monitoring (Prometheus, Grafana, Alertmanager)
Compute01	8	64 GB	Basic 200 GB Disk for TiDB	TiDB DB Engine PD
Storage01	8	32 GB	Basic 200 GB Disk for TiDB	TiKV
Storage02	8	32 GB	Basic 200 GB Disk for TiDB	TiKV
Storage03	8	32 GB	Basic 200 GB Disk for TiDB	TiKV

Table 5.1: Test Cluster Setup

is v7.2.0. Our implementation is based on TiDB v6.7.0-alpha. The roles assigned to machines, regarding to TiDB and test execution, are listed in Table 5.1. All machines, except of Monitor01, are dedicated solely to the TiDB cluster. Monitor01 manages the cluster, contains a clone of our source code repository, used for building new versions and patching the database, and manages and runs the experiments.

By default, the memory quota for a query in TiDB, is set to 1 GiB. This gives every query a possible amount of memory of up to this quota. If this quota is exceeded, the database actively tries to stop the corresponding query. [Pin23l] We quickly realized that we are not going to be very effective with this low quota. Therefore, we raised the quota multiple times, until finally reaching 56 GiB. The quota can be set with an SQL client through the following command: `SET GLOBAL tidb_mem_quota_query = 56 << 30;`, and checked with `show global variables like 'tidb_mem_quota_query';`. The respective value for our cluster after setting the quota is 60129542144 bytes, which is equivalent to 56 GiB.

To limit the time spent for testing, and keep the runtimes at a reasonable level, we set the timeout for a query to 15 minutes. This can be done with the optimizer hint `MAX_EXECUTION_TIME([durationInMS])`. If a query takes longer than the given time, the database actively tries to stop the query. In case this is not possible, which happens from time to time, our testing scripts manually restart the database by issuing the `shutdown` command. TiUP automatically restarts the database again.

Figure 5.1 illustrates the flow of our tests. In this illustration, the database is assumed to be setup, and fully initialized with test data. The left part of the figure shows the

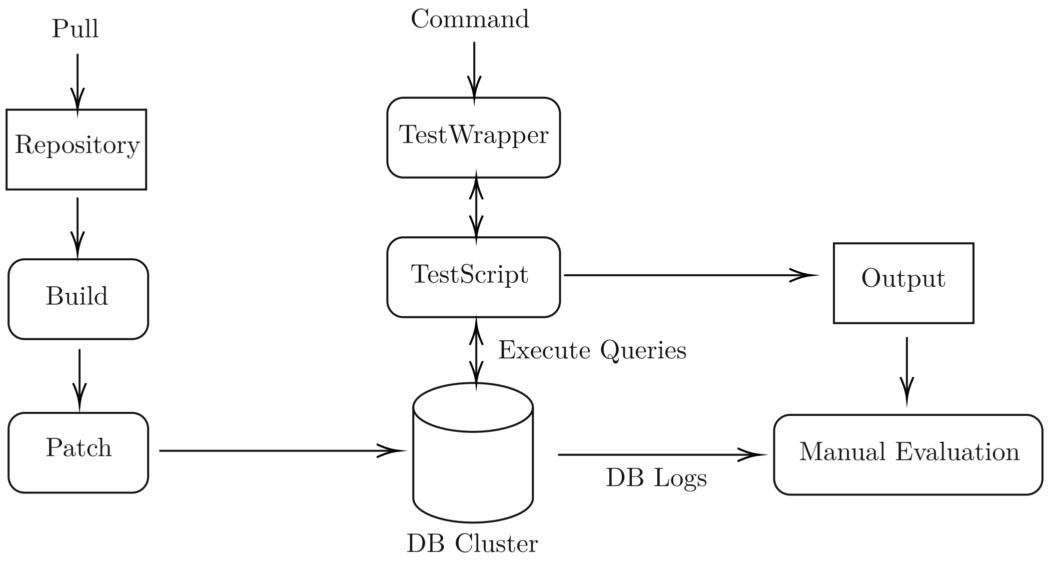


Figure 5.1: Test Flow

process of patching the database system with a new TiDB database engine. To do that, we pull the version to a local repository, where the build process is initialized. The built and prepared binary can then be used to patch the database cluster.
Tests are executed through a command to either our test wrapper script, or our test script, both written in python. The difference between them is that the test wrapper detects failures and queries running longer than allowed, in case the database system fails to automatically stop a query. We run queries either with `explain` to just create a plan, or with `explain analyze`, to also execute the plan. Additionally, for 0MA queries, we run the plain `select` statement for validation of the returned data. Other queries are validated through the number of result tuples returned. The output data, including runtimes, `explain`, `explain analyze`, and validation outputs is stored. Together with the database logs, which contain further data about memory consumption of queries, etc., the output is then manually, respectively semi-automated, evaluated. All elements shown in the illustration take place, or are located on Monitor01, except the database cluster and the final (manual) evaluation.

5.1.2 Testdata

For evaluating our implementations with queries, we need some data. The requirements to our test data are that the schema is comprised of a medium to high number of relations, and that the amount of data is big enough to be able to recognize situations in which intermediate results of queries are getting too big to compute them. We found a good test dataset which complies with our requirements, the MusicBrainz database [MF23]. This dataset has already been used in previous works of Gottlob et al. [GLL$^+$23], working with structure guided query optimization, and Mancini et al. [MKC$^+$22], working on parallel join computation. The MusicBrainz database consists of 232 relations, with 55 relations relevant for our tests, ending up with 30 - 40 GiB of data in each of our TiKV nodes, including redundancy among the nodes.

We created the schema and inserted the data once on our cluster, working on the same data for the whole time of experimentation.

5.1.3 Queries

Mancini et al. [MKC+22] also provided queries for the MusicBrainz database, ranging from 2 to 30 involved relations. 84 out of 435 queries are of hypertree width 2, meaning cyclic queries, all others are acyclic, with hypertree width 1. We use two sets of queries, first, full enumeration queries, essentially selecting all columns of the result with `SELECT * FROM ...`, and 0MA queries used in the previously mentioned paper by Gottlob et al. [GLL+23], using functions like min and max to compute an aggregate over the output. For some of our tests, we may only use a subset of the queries. Our tests include running the aforementioned queries sequentially, one at a time. We refer to the execution time of the query plans' root executor as query runtime.

5.2 Feature Evaluations

In the following section, we will discuss the evaluations for our additional features, and assess their performance and usefulness.

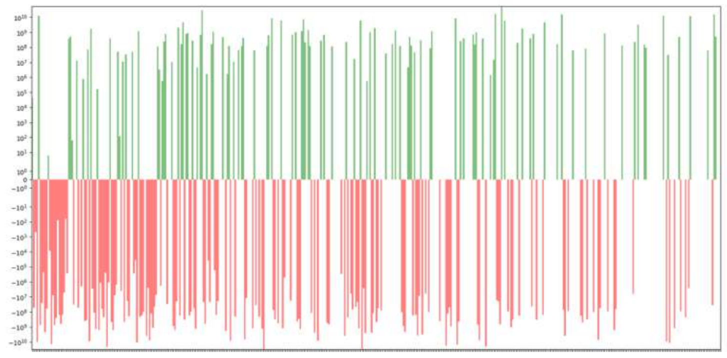
5.2.1 CTE Mock

For the tests of CTEMock, we ran all queries on the basic implementation and the basic implementation including the CTEMock feature. The full enumeration and 0MA queries were run with separate planning (`explain`) and execution (`explain analyze`) runs. We conducted one run per implementation.

We expected the results to show a lower memory usage with CTEMock for the majority of the queries. Additionally, as the memory usage is expected to be lower, we also expected CTEMock to be able to run more queries than the basic version (without exceeding the memory quota).

Results

The results more or less reflect our expectations for the CTEMock feature. As we can see in Figure 5.2, where each line on the x axis shows one query, and the green or red line show an advantage in form of a positive difference in regards of memory usage to CTEMock (red) or basic (green), the feature benefits 0MA queries (244 queries better with CTEMock vs. 59 queries better w/o CTEMock) more than full enumeration queries (153 vs. 97). Other than expected, this does not reflect on the number of unsolved


(a) Memory Difference Plot Full Enumeration

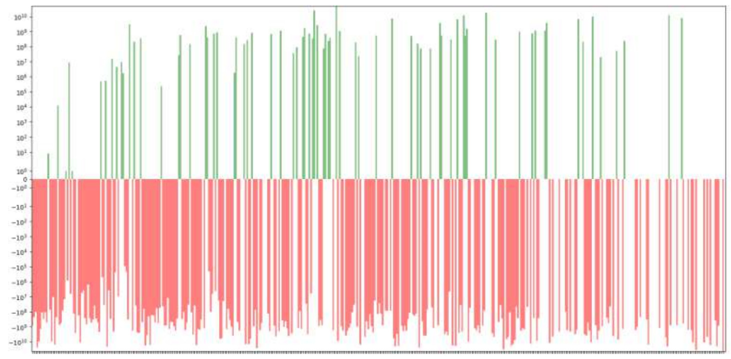

(b) Memory Difference Plot 0MA

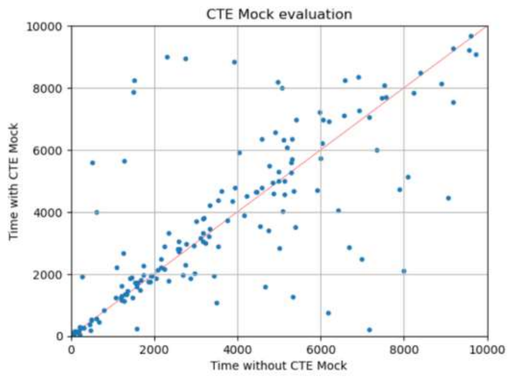
Figure 5.2: CTEMock Evaluation: Memory Difference Plots. Green: w/o CTEMock better, Red: with CTEMock better

instances, as CTEMock solves the same number of instances as basic in full enumeration, and three instances less than basic with 0MA queries.

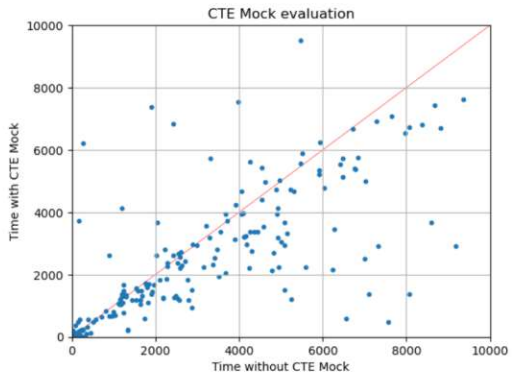
Interestingly, in terms of runtime, full enumeration without CTEMock is better for 153 instances, compared to 91 with CTEMock. 0MA queries turn the results, with CTEMock being faster for 196 instances, compared to 107 without CTEMock. Additionally, this can be clearly seen in the corresponding plots in Figure 5.3.

In both aspects, memory usage and runtime, we saw a better result for CTEMock with 0MA queries. We think this is due to the fact that 0MA queries only use CTEMocks, other than full enumeration queries, where CTEMocks play a minor role regarding the amount of data being handled by them. With 0MA queries, the whole result for all hypertree decomposition nodes, especially the root's result, which would otherwise need to be persisted in case of full enumeration queries, is handled by CTEMocks.

With the results in mind, we conclude that CTEMocks are an effective measure to reduce resource usage through CTEs in our implementation of structure guided query execution.



(a) Runtime Scatterplot Full Enumeration, Excerpt



(b) Runtime Scatterplot 0MA Queries, Excerpt

Figure 5.3: CTEMock Evaluation: Runtime Scatterplots. Above Red Line: w/o CTEMock better, Below: with CTEMock better

5.2.2 Join Costs

Early in our testing process, we saw that the quality of hypertree decompositions heavily varies. Also, for cyclic queries, there were a lot of plans including cartesian products, thus being potentially inefficient. The joinCosts feature was built to fix these aspects. For evaluating the feature, we conducted 5 runs of just plan creation of all full enumeration queries for the joinCosts feature and the basic implementation. Additionally, we ran all full enumeration queries for these two versions.

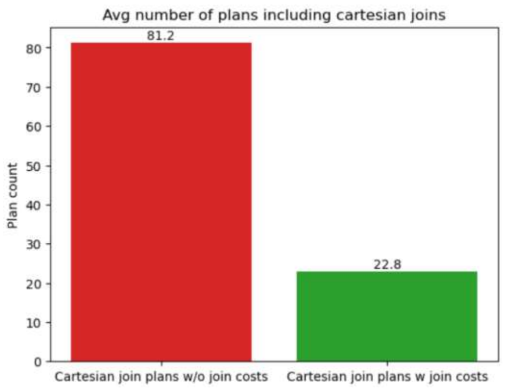
Our expectations were the elimination of a majority of the plans with cartesian joins. Moreover, due to better quality of hypertree decompositions, we expected lower runtimes, and, with the two mentioned effects combined, more solved instances.

Results

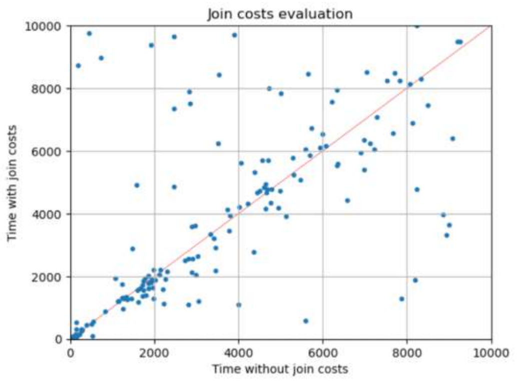
Regarding the planning of all queries, our results clearly show an improvement, lowering the average number of cartesian products per run by almost 72% across all full enumeration queries, with the joinCosts feature, as illustrated in Figure 5.4a. Our expectations were met in this regard.

As visible in Figure 5.4b, the majority of datapoints lies underneath the red line, indicating a runtime advantage for the joinCosts feature. Looking at the exact values of queries solved by both versions, joinCosts has a clear advantage with 130 queries faster with joinCosts vs. 116 queries faster with the basic version. Also, while the basic version has 173 unsolved instances, the joinCosts feature just fails with 139 instances.

The situation is even more clear for cyclic queries. Of 84 cyclic queries included in our test queries, 80 are unsolved with the basic implementation, however, only 45 with the joinCosts feature, a reduction of over 40%.



(a) Average Number of Plans including Cartesian Products over 5 Planning Runs



(b) Join Costs Evaluation: Runtime Scatterplot, Excerpt

Figure 5.4: Join Costs Evaluation: Number of Plans with Cartesian Products and Runtime Scatterplot

The results clearly show the benefits of the joinCosts feature, additionally to the fact that a similar idea was already used by Gottlob et al. [GLL+23]. Hence, this feature will be included in the final implementation.

5.2.3 Early Stop

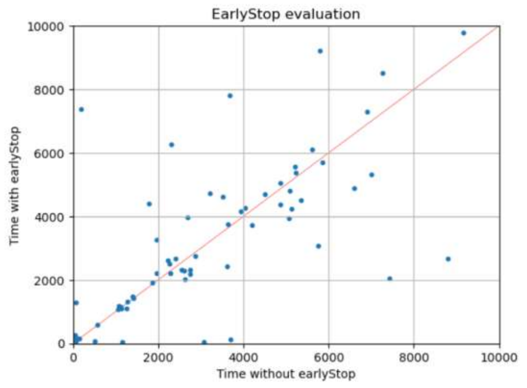
Queries with an empty result are especially well represented in our test queries. Specifically, the queries with a higher number of involved relations often do not have tuples in their results due to the many conditions applying to them. To help improve these cases, we implemented the earlyStop feature.

For evaluating the effects of this feature, we conducted a test run of about a third of our full enumeration test queries, mostly with non-empty results, to check if there is an impact on these. Besides, we ran all full enumeration queries with empty results, to see if there are any benefits caused by the newly added feature. The resulting data is compared with the basic implementation.

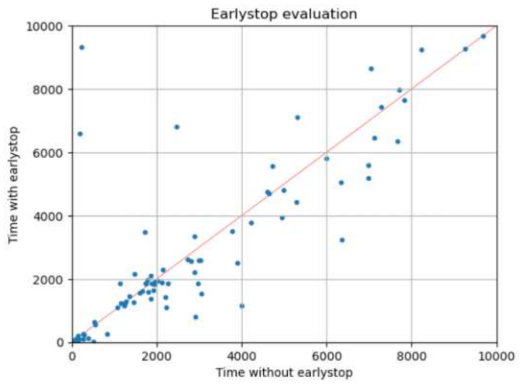
We are looking for an improvement of runtimes and resource usage (in particular memory usage) for queries with empty results through the feature, along with no negative impact on other queries.

Results

This time, the result for the empty result queries was worse than our expectations. Especially in terms of runtime, the earlyStop version and the basic version had a tie in number of instances where one of each was better. This is clearly visible in the distribution of datapoints in Figure 5.5a. Nonetheless, a look at the average runtime difference of these queries shows us a 25 second benefit for earlyStop, with a 40 ms



(a) Empty Result Queries Runtime Scatterplot, Excerpt



(b) All Queries Runtime Scatterplot, Excerpt

Figure 5.5: Early Stop Evaluation: Runtime Scatterplots

median advantage. EarlyStop and the basic version solved nearly the same number of instances, with an advantage of one instance for the basic version. In terms of memory usage, there is a clear benefit on the earlyStop version, with 57 instances using less memory than basic, in comparison to 40 instances using more memory than basic. More so when inspecting the average of memory differences, for which earlyStop uses 1.1 GB less memory than the basic version, along with the median testifying earlyStop uses 26 MB less memory per query.

The check for other queries shows no visible pitfall whatsoever. There is even an improvement without any changes in the execution flow of these queries, also visible in Figure 5.5b. We explain this improvement with the variation of hypertree decomposition quality, and a coincidence in accumulation of better hypertree decompositions.

Following, although the results were not as clear as expected, we see earlyStop as a notable addition to our basic version. Especially as there is no drawback for queries with a non-empty result. Also, we should keep in mind that the effect of the earlyStop feature is dependent on the depth of the first empty result in the query plan, as well as the point in time (empty result occurs quickly vs. empty result occurs later), at which the empty intermediate result is detected. Depending on these factors, the impact could vary in each run, from a significant improvement to no improvement at all. However, we see no case where worse results, solely caused through this feature, could happen.

5.2.4 Secondary Plans

The secondaryPlans feature is meant to further increase the quality of query plans. We evaluated this feature along with the basic version, as well as in combination with the joinCosts feature. This is because the joinCosts feature and this feature have the same overall goal of increasing the query plan quality. Therefore, it is possible that the extent and measure of benefits is affected by the combination of these two features.

We introduced a way to measure the effect of secondaryPlans with this feature, by logging the initial cost of the physical plan, as well as the final cost of the physical plan. In this way, it is possible to measure to which extent the query plan improved through the secondaryPlans feature.

The method of testing for this evaluation contained 5 runs of creation of plans for all full enumeration queries with the secondaryPlans feature, and combined with the joinCosts feature. With this data, percentage improvements for these two versions, compared to the initial plan computed with the basic version, or in the combined version, compared to the joinCosts feature, can be recorded.

Additionally, we chose 4 full enumeration queries with 4, 6, 11, and 30 involved relations, one of which is a cyclic query (11ag), to directly compare the execution with the basic version, secondaryPlans, and secondaryPlans combined with joinCosts, over 5 runs.

We expected to see improvements in query plan costs by the secondaryPlans feature, simply because with more tries and differing hypertree decompositions, there is a higher probability of computing a better plan. Moreover, combined with the joinCosts feature, it is possible that the improvements are lower than without this feature. Due to the fact that, when including the joinCosts feature, more consistent hypertree decompositions are produced, the probability for bigger differences in plan costs is less.

Results

The results of the test show the effects of the two additional features secondaryPlans and secondaryPlans combined with joinCosts. Since the queries gain complexity, and therefore their hypertree decompositions are more complex, including more possible results, with a rising number of involved relations, we separated the queries in three equal sized groups, from a list of queries ordered by rising number of relations, to reflect their complexity (i.e., Group 1 $\sim$ 2-10 relations, Group 2 $\sim$ 11-20 relations, Group 3 $\sim$ 21-30 relations). Each group has a higher query complexity than the previous one. The formula for calculating the percentage improvements is as follows: $(1 - \frac{costafter}{costbefore}) * 100$.

In Table 5.2, we can see that with rising number of involved relations, a higher rate of improvement is achieved. Furthermore, the discrepancy between averages and medians indicates outliers on the upper range of values, with the majority of values being in the lower range. Reviewing the boxplots of the distributions confirms the situation.

As expected, the improvement of plans for the combination of the joinCosts and secondaryPlans features is lower than without joinCosts. Also, the range of results from the runs indicate a slightly lower variation. The additional planning time, as listed in Table 5.3, is consistent between these two versions, and increases with rising query complexity.

The single queries executed generally show a visible difference between the runtime with the basic version and the other two versions in Figure 5.6. Especially Figure 5.6c, for the cyclic query 11ag, illustrates the difference in quality of the hypertree decompositions achieved through multiple tries with secondaryPlans, where 4 out of 5 executions timeout, compared to the combined version in which only 2 out of 5 executions run into the timeout. Moreover, the basic version runs into timeout on every execution. As observed before, improvements between the basic version and the other versions are higher for more complex queries.

Besides, we can clearly see that even with our improvements, bad hypertree decompositions, and therefore bad query execution plans with long runtimes, occur. Examples are the red execution by the secondaryPlans version for query 4ag, or the timeout by the combined version for query 30ag.

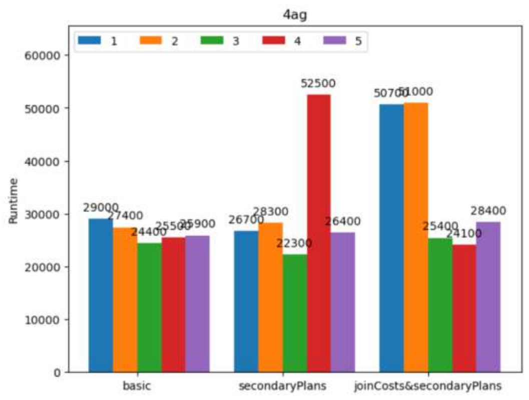
Concluding this evaluation, we see the improvements for the used runtime as worthwhile, in particular considering the improvements for cyclic queries, and queries of the second and third group. Even if the combination of the two aforementioned features lowers the effect of secondaryPlans, it is definitely beneficial to integrate these two additions into the final version of our implementation.

	secondaryPlans	joinCosts&secondaryPlans
Group 1 Avg.	6 - 8 %	6 - 8 %
Group 1 Med.	$\sim$ 0 %	0 %
Group 2 Avg.	25 - 33 %	20 - 26 %
Group 2 Med.	2 - 15 %	4 - 6 %
Group 3 Avg.	39 - 51 %	30 - 37 %
Group 3 Med.	10 - 47 %	8 - 18 %

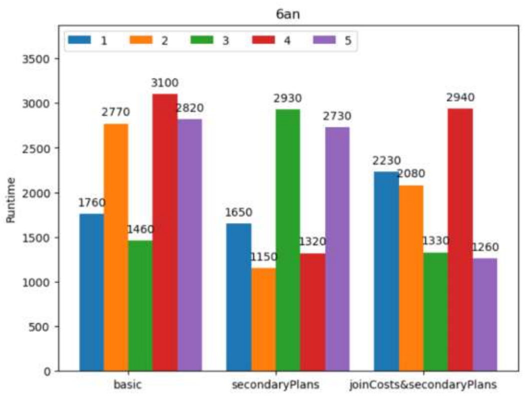
Table 5.2: Secondary Plans & Join Costs Evaluation: Improvements (%) over Init Plan

	secondaryPlans	joinCosts&secondaryPlans
Group 1 Avg. PT	5 - 6 ms	5 - 6 ms
Group 1 Med. PT	5 - 6 ms	5 - 6 ms
Group 2 Avg. PT	18 - 20 ms	18 - 19 ms
Group 2 Med. PT	16 - 18 ms	17 - 18 ms
Group 3 Avg. PT	42 - 79 ms	43 - 44 ms
Group 3 Med. PT	38 - 41 ms	38 - 41 ms

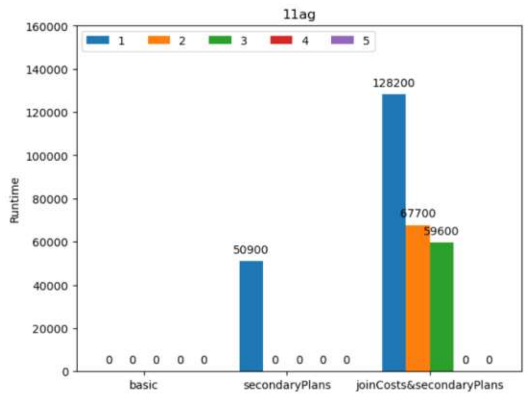
Table 5.3: Secondary Plans & Join Costs Evaluation: Additional Planning Time (ms)



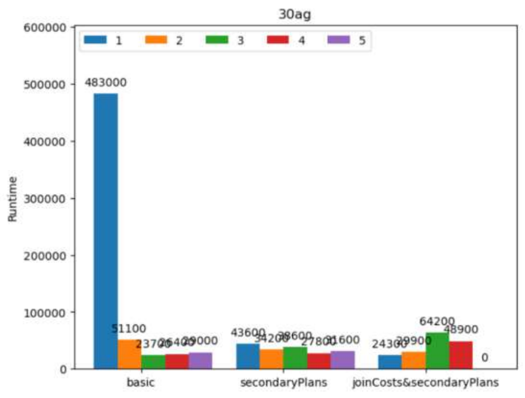
(a) Query 04ag, Full Enumeration



(b) Query 06an, Full Enumeration



(c) Query 11ag, Full Enumeration



(d) Query 30ag, Full Enumeration

Figure 5.6: Secondary Plans & Join Costs Evaluation: Runtime (ms) over 5 Runs

5.2.5 Summary

Summarized over all feature evaluations, we can conclude that all features have their benefits, making their integration into the final version irresistible. Even considering the downsides, which especially occur because of the combination of secondaryPlans and joinCosts do not overrule this decision. Consequently, we will integrate all four optimizations, CTEMock, joinCosts, earlyStop, and secondaryPlans, into the final implementation, which is then considered for the main evaluation.

5.3 Main Evaluation

The evaluation of the final version was conducted in multiple runs for our implementation, and one run for normal query evaluation, since there are no differing plans, and therefore no fluctuations in query runtime, expected. Among the runs, we chose the result with the respective lowest runtime as base for our evaluations. We ran the full enumeration-, and 0MA queries, acyclic- and cyclic queries, for both, the normal query evaluation, and our final implementation.

Our expectations included some improvements through structure guided query optimization and evaluation in terms of runtime and solved instances. As we already discussed in Section 4.2 on page 48, the CTEs used in the plans result in increased memory usage.

5.3.1 Results

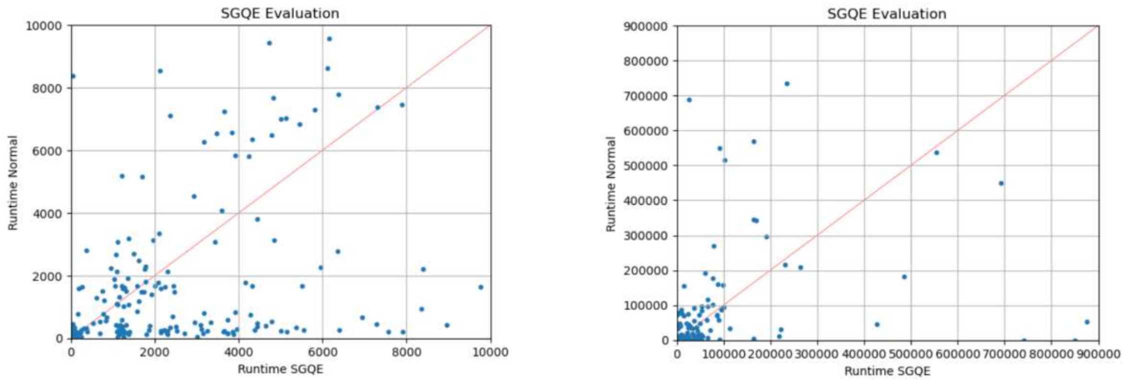
The overall results show a benefit for structure guided query optimization for some queries, confirming the advantages of this approach, compared to normal query evaluation. The results are better for full enumeration queries than for 0MA queries, despite the need for only one phase of Yannakakis' algorithm with 0MA queries.

Full Enumeration

As mentioned, we see an improvement in runtime for some queries, however, the majority of queries still evaluates with a faster runtime through the original version of the database system, with 125 instances being solved faster through structure guided query evaluation, compared to 180 with the original evaluation and optimization, as can be seen in Figure 5.7. Figure 5.8 presents the extent of runtime differences between the two versions. A bar going all the way to the top or bottom of the diagram represents a query solved by one of the two implementations. Missing bars show queries for which none of the versions was able to calculate a solution before running into the timeout, or out of memory. Our implementation also holds a little advantage of one query in instances solved.

As expected, the memory usage is higher in our implementation for a majority of the queries, where 274 instances use less memory when evaluated with normal query evaluation, compared to 32 with our implementation. On average, the original implementation uses 2.8 GB less memory when evaluating one of the queries. This one-sided relation is also illustrated in Figure 5.9. Other than we would have expected, instances with the biggest memory benefit for our implementation are not characterized by exceptionally high memory usage when evaluated with the normal version.

Cyclic and acyclic queries replicate the situation with 30-45% of the instances being faster with our implementation when both versions solved them, and the benefit for the majority on the side of the original version (14 vs. 27 instances for cyclic queries, 111 vs. 153 instances for acyclic queries). The memory benefit remains as in the overall picture, with an advantage towards the original implementation. Considering the number of solved instances, our implementation still holds a favor of three instances over normal



(a) Runtimes up to 10 Seconds (b) Runtimes up to Timeout (900 Seconds)

Figure 5.7: Final Evaluation: Full Enumeration Overall Runtime Scatterplot

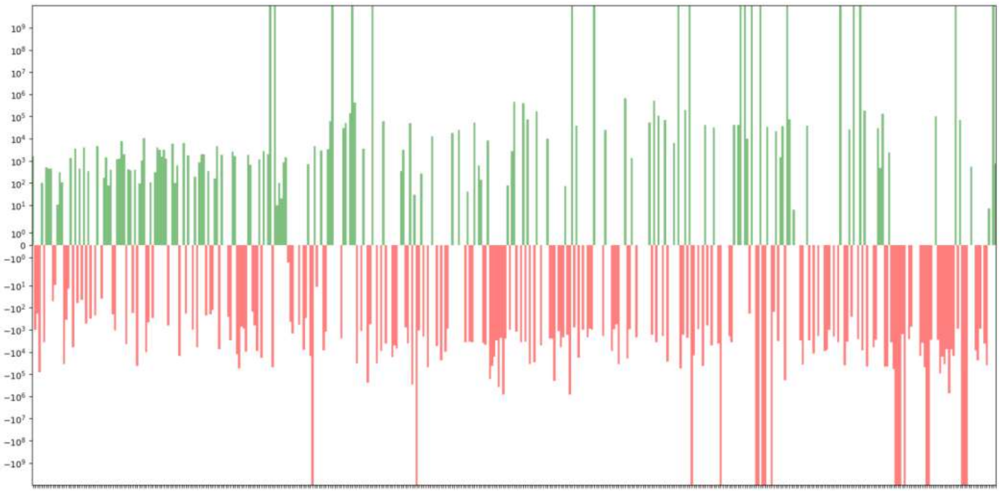


Figure 5.8: Final Evaluation: Full Enumeration Runtime Differences (ms); Red Shows an Advantage for the Original Version, Green for our Implementation; one Bar per Query

evaluation and optimization for acyclic queries. However, looking at cyclic queries, our implementation solves two instances less.

To identify any correlations of the results with the size and complexity of the queries, we separated our test queries into 10 roughly equal sized groups, created from a continuous list of queries. The results are illustrated in Table 5.4. Regarding runtime, our implementation prevails in the first two groups, with additional close results in groups 3 and 7. The memory counts repeat the data from before, testifying our implementation a disadvantage. In the matter of solved instances, the data shows a tie. Although this characterization into groups shows us that there is a potential advantage for our implementation for smaller queries, there are other factors, like cyclicity, which first appears in a 09xx query, and is larger represented in higher groups, and queries with an empty result, which we

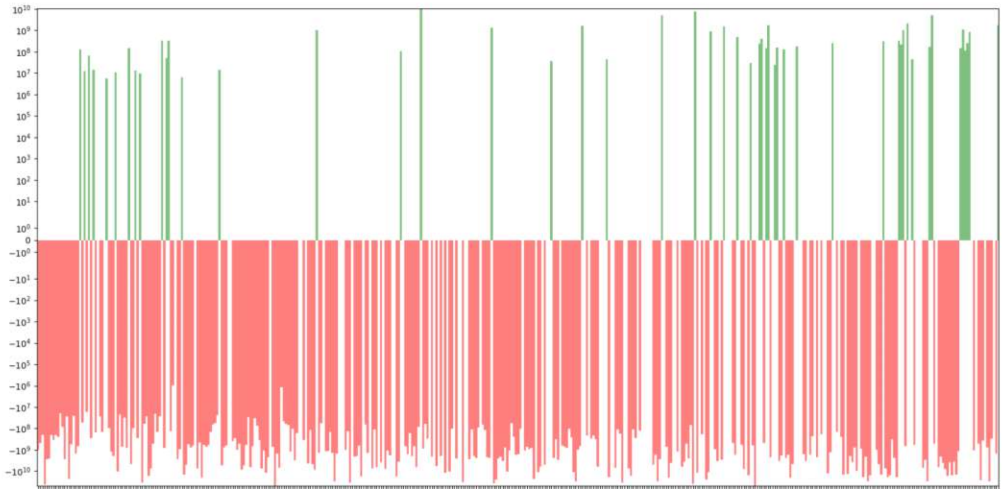


Figure 5.9: Final Evaluation: Full Enumeration Memory Usage Differences; Red Shows an Advantage for the Original Version, Green for our Implementation; one Bar per Query

Group	Queries	Runtime count (SGQE vs. Normal)	Memory count (SGQE vs. Normal)	Unsolved instances (SGQE vs. Normal)
0	02aa - 04an	23 vs. 19	7 vs. 35	2 vs. 2
1	04ao - 07am	25 vs. 14	7 vs. 33	4 vs. 4
2	07an - 10al	15 vs. 21	0 vs. 36	6 vs. 7
3	10am - 13ak	13 vs. 16	1 vs. 28	12 vs. 14
4	13al - 16aj	11 vs. 21	1 vs. 31	12 vs. 12
5	16ak - 19ah	7 vs. 22	3 vs. 26	12 vs. 14
6	19ai - 22af	9 vs. 16	2 vs. 23	16 vs. 17
7	22ag - 25ad	10 vs. 10	5 vs. 15	18 vs. 17
8	25ae - 28ab	7 vs. 20	2 vs. 25	13 vs. 13
9	28ac - 30ao	5 vs. 21	4 vs. 22	15 vs. 11

Table 5.4: Final Evaluation: Full Enumeration Groups and Results

will take a look at next.

These queries with an empty result, which we name zero result queries, are prominently represented in our test database, as we identified over 150 instances meeting this requirement. We filtered the results to take a look at these queries. Thus, we saw that our implementation performs worse with these queries than overall. The runtime is better with normal query evaluation for 102 queries, whereas our implementation dominates in 28 cases, as illustrated in Figure 5.10. Additionally, structure guided query optimization and evaluation has 16 instances unsolved, with only 5 for the original implementation.

On the other hand, non-zero result queries, queries whose result is not empty, of which over 280 are contained in our test queries, seems to deliver a better result for our implementation. It has better runtime for the majority of these queries (97 vs. 78), as

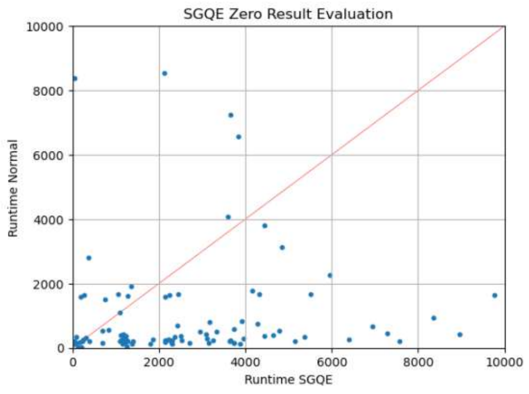
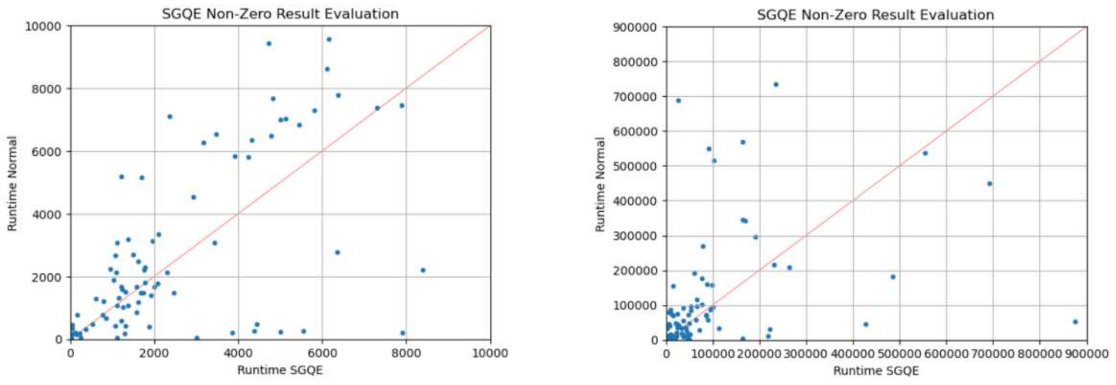


Figure 5.10: Final Evaluation: Full Enumeration Zero Result Queries Runtimes Scatterplot, Relevant Part



(a) Runtimes up to 10 Seconds (b) Runtimes up to Timeout (900 Seconds)

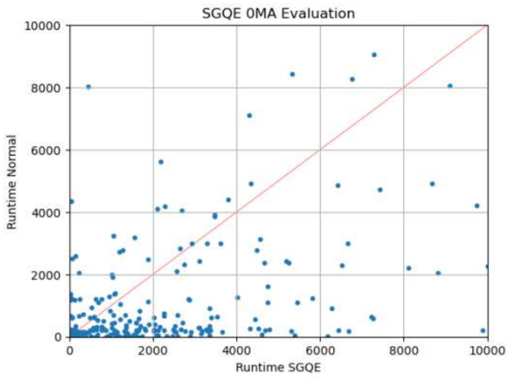
Figure 5.11: Final Evaluation: Full Enumeration Non-Zero Result Queries Runtime Scatterplot

shown in Figure 5.11, and solves 12 instances (11%) more, than normal query evaluation.

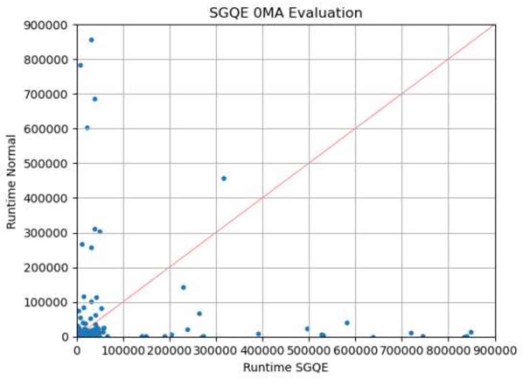
Following this result, we had the assumption that big results suit our implementation better than small or empty results. We could check this assumption with the 20 queries with most and least (i.e., all zero result queries) result tuples. Other than expected, both groups show a benefit for the original query evaluation.

0MA

The overall situation with 0MA queries is similar as with full enumeration queries. Our implementation has a better runtime for some instances. However, the majority of instances still evaluates faster with the original version of the database system. Our data shows a runtime advantage in 250 instances for the conventional optimizer and evaluation, as opposed to 106 instances for structure guided query optimization and evaluation. In



(a) Runtimes up to 10 Seconds



(b) Runtimes up to Timeout (900 Seconds)

Figure 5.12: Final Evaluation: 0MA Overall Runtime Scatterplot

the runtime diagrams in Figure 5.12, we can see that the results are densely packed parallel to the x axis, with an advantage for normal evaluation. Also, there are some outliers further along the x and y axis, representing exceptionally good results for one of the two versions.

Regarding the memory usage, although for 0MA queries, only the first phase of Yannakakis' algorithm is needed, normal evaluation has a big lead of 344 instances using less memory than our implementation, compared to 13 instances for structure guided query evaluation. Following, on average, an instance uses 2.3 GB less memory with the conventional evaluation than with our implementation.

A statistic which gives our version an advantage is the number of solved instances. It manages to solve 26 instances (45%) more than the original version.

As with full enumeration queries, the statistics continue as before regarding the groups of cyclic and acyclic queries, evaluated on their own. Our data shows that cyclic queries are handled worse by our implementation, with 8 vs. 41 instances having lower runtimes with structure guided query evaluation vs. the original query evaluator. Acyclic queries deliver a slightly better ratio with 98 vs. 209 instances in favor of the respective version. Interestingly, our implementation seems to be more performant in handling hard instances of acyclic queries (60% less unsolved instances than normal evaluation), compared to hard instances of cyclic queries (20% less unsolved instances).

We proceeded to separate the queries into 10 groups, matching with previous groups created for full enumeration queries. The runtime counts in Table 5.5 show fluctuating results. Apart from the weaker results for structure guided query execution for the last two groups, there is no dominating trend towards one or the other version. In terms of unsolved instances however, there is an advantage for our implementation in the matter of the trend of unsolved instances. For the conventional version, the number rises quicker than for our version, indicating an advantage for structure guided query execution with rising query size.

Group	Queries	Runtime count (SGQE vs. Normal)	Memory count (SGQE vs. Normal)	Unsolved instances (SGQE vs. Normal)
0	02aa - 04an	13 vs. 31	1 vs. 43	0 vs. 0
1	04ao - 07am	11 vs. 32	2 vs. 41	1 vs. 0
2	07an - 10al	12 vs. 30	1 vs. 41	1 vs. 1
3	10am - 13ak	19 vs. 20	0 vs. 39	3 vs. 3
4	13al - 16aj	11 vs. 28	1 vs. 38	2 vs. 4
5	16ak - 19ah	12 vs. 25	3 vs. 34	1 vs. 6
6	19ai - 22af	8 vs. 20	2 vs. 26	5 vs. 12
7	22ag - 25ad	10 vs. 15	0 vs. 25	9 vs. 11
8	25ae - 28ab	5 vs. 25	1 vs. 30	4 vs. 9
9	28ac - 30ao	5 vs. 24	2 vs. 27	5 vs. 11

Table 5.5: Final Evaluation: 0MA Groups and Results

0MA queries previously identified as zero result queries produce a better percentage of instances with lower runtimes for our implementation, than with full enumeration queries, with 41 instances for our implementation vs. 92 instances for the original evaluation being solved faster than the respective other version. Yet, we still see more unsolved instances, with 14 vs. 6, with structure guided query evaluation.

Whereas the full enumeration result showed a benefit for our implementation for non-zero result queries, this is not true for 0MA queries, where our implementation is better for only 30% of the queries.

5.4 Discussion

Now, we are going to discuss and analyze the aforementioned results. First, we focus on the memory usage. The increased memory usage by our implementation is the consequence of temporarily storing intermediate results for the execution of Yannakakis' algorithm. Moreover, we see that in 0MA queries, our implementation is not as far off of the conventional evaluation as with full enumeration queries, on average 2.3 GB vs. 2.8 GB. This is a direct impact of the reduced complexity of plans created for 0MA queries, which only contain the first phase of Yannakakis' algorithm. Also, we assume this as the reason for the lower number of unsolved instances by structure guided query execution, compared to the original implementation, for 0MA queries, as opposed to full enumeration queries.

As we saw in the test results, 0MA queries are handled more performantly by the original version, than full enumeration queries. Therefore, the ratio where our implementation is performing better is higher with full enumeration queries (40%), than with 0MA queries (30%).

One reason for this behavior could be the usage of DistSQL operations via coprocessor on the distributed storage, which allow some computations to be performed on the storage nodes. One of these operations is a selection filter when fetching data via a Table- or IndexReader. As we found out when reviewing some query plans, the DBMS already tries to implement an approach to reduce the number of redundant tuples, only fetching matching rows for the current join operation. This is especially useful in combination with various types of IndexJoins, in which only matching index entries are read from the storage. Thus, there is no redundant computation on this immediate join, since every fetched tuple matches.

Our implementation, however, rarely uses coprocessor functions besides simple fetching. Normally, the physical optimizer should automatically recognize these possibilities. However, this is not always the case. As a result, some operations, for instance when fetching big relations, add a noticeable delay to query evaluation.

Figure 5.13 illustrates an execution plan created by the conventional optimizer and evaluator, heavily using coprocessor tasks (marked red in the illustration). The two leftmost and deepest coprocessor tasks do not contain any additional condition. The remaining ones, however, include equality conditions adopted from the above joins. Subsequently, they only return matching tuples for the current joins. In particular, the join with the recording relation, which contains around 30 million tuples, is sped up significantly through this measure, reducing the number of fetched tuples to roughly 372000.

Additionally, plans created by the original implementation are oriented on the indexes of the database, which is not the case for our implementation, realizing the plan implied by the hypertree decomposition. This does not rule out usage of indexes at all, they are used as soon as it is possible to use them. However, with structure guided query optimization, the plan is not primarily built to optimally utilize all possible indexes, since the hypertree decomposition specifies the join order without the knowledge of existing indexes.

We saw a huge variation of up to multiple orders of magnitude in runtime of queries run with structure guided query optimization between different runs. We think this is primarily due to the varying quality of hypertree decompositions. Quality of hypertree decompositions, in our view, is defined by the ability of the implied query plan to reduce the number of redundant intermediate result tuples as quickly as possible. Additionally, the runtime fluctuations for queries with empty results can be explained with the differing effect of the earlyStop feature. Query plans produced by structure guided query optimization and execution contain more steps than conventional query plans, causing a higher runtime. Additionally, the earlier an empty intermediate result is detected by the earlyStop feature, the earlier the query finishes. However, if this happens at a later point, the query ultimately runs longer. The point of detection is primarily determined by the hypertree decomposition. Hence, differing hypertree decompositions yield different query runtimes.

Figure 5.14 illustrates two plans for an instance giving structure guided query evaluation the better runtime. On the left side, the plan created for the third phase of Yannakakis' algorithm by our implementation is shown, on the right side the conventional plan. CTEs,

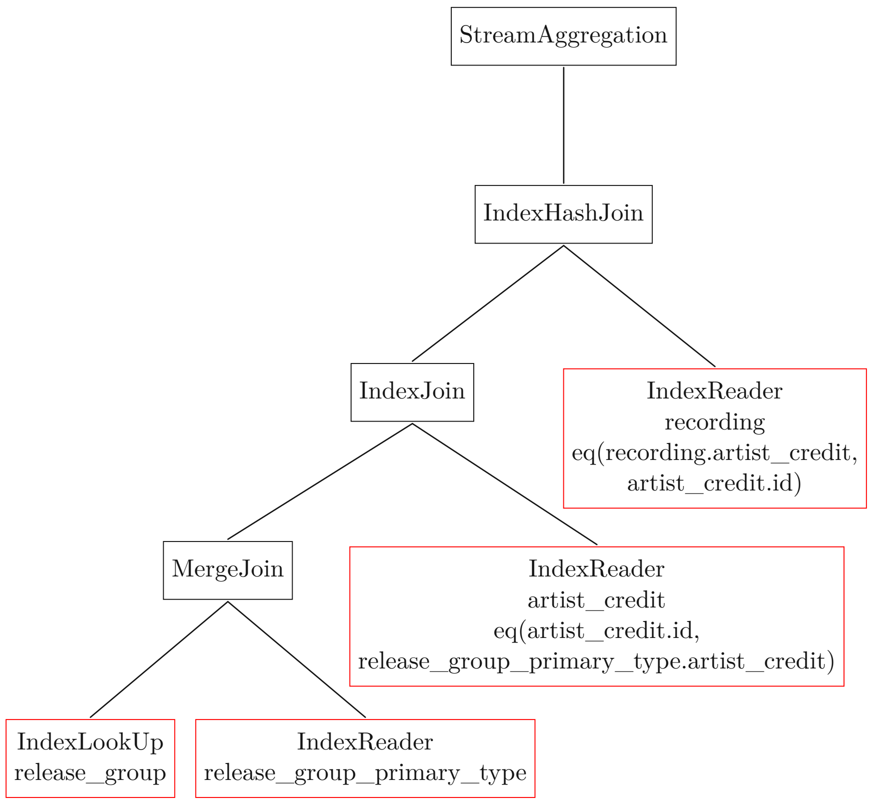
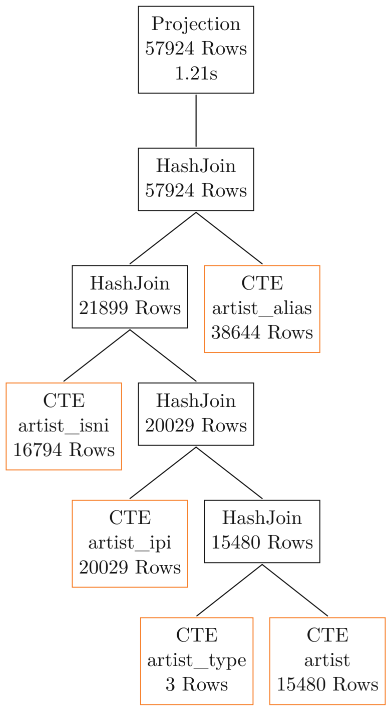

StreamAggregation
IndexHashJoin
IndexJoin
IndexReader recording eq(recording.artist_credit, artist_credit.id)
MergeJoin
IndexReader artist_credit eq(artist_credit.id, release_group_primary_type.artist_credit)
IndexLookUp release_group
IndexReader release_group_primary_type

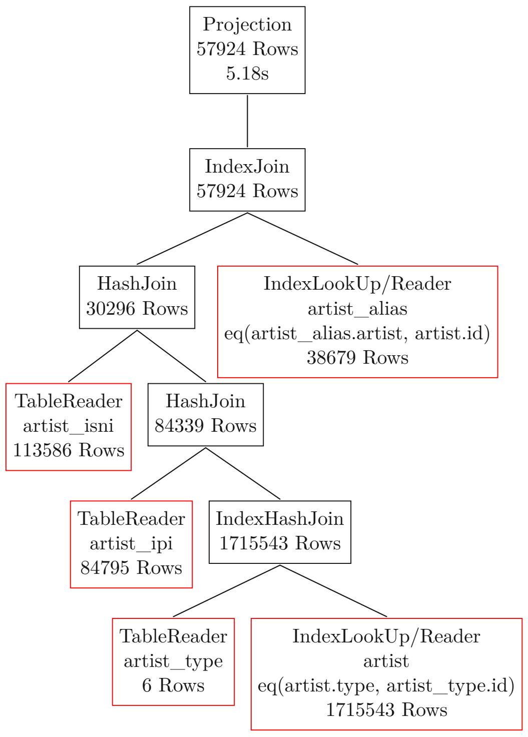
Figure 5.13: Execution Plan of 04aa as 0MA Query with the Original Version

plan nodes with a subtree underneath them, are outlined in orange, coprocessor tasks in red. As we can see, the number of rows in the CTE nodes, which already passed through the semi-join stages, are far less than on the conventional plan. Although the conventional plan uses selections through coprocessor tasks twice, it is not possible to achieve the same effect as with structure guided query evaluation, since coprocessor selections do only optimize locally, for the current join. Albeit our implementation added the semi-join stages, the final runtime is far lower than with the conventional plan, because of less tuples to be handled during the inner joins.

In conclusion, it is clearly observable that there are cases, distributed over all types of queries, in which structure guided query optimization and evaluation show their advantages and outperform conventional query execution. Moreover, there is a class of queries performing better than others. In particular, full enumeration queries, whose result is expected to be non-empty, are the best performing group of queries, with a benefit for structure guided query evaluation in a majority of instances, according to our evaluation. Additionally, the number of failed instances for 0MA queries were significantly reduced. However, we should keep in mind that there are still big fluctuations in query execution times from run to run. Therefore, sensible usage of this technique should always consider the expected plan costs, compared to conventional plans, to get the



(a) Execution Plan for the third phase of Yannakakis' Algorithm of 05al as Full Enumeration Query created with our Implementation



(b) Execution Plan of 05al as Full Enumeration Query created with the Original Version

Figure 5.14: Comparing two plans: 05al by the Original Version vs. Our Implementation

biggest benefit possible and exclude worse query plans. Nevertheless, structure guided query optimization, as we implemented it, is a notable addition to an existing query optimizer, resulting in a lower runtime for some queries.

CHAPTER 6

Conclusion

In this thesis, we have investigated how structure guided query optimization can be used to reduce query runtimes and minimize the number of unsolved queries through memory exhaustion or timeouts. Furthermore, we presented our implementation realizing structure guided query optimization and evaluation through Yannakakis' algorithm in the NewSQL DBMS TiDB for acyclic and cyclic queries, with a hypertree width up to 2 (i.e., almost acyclic queries). Our implementation manages to execute all stages of Yannakakis' algorithm in one statement through usage of CTEs. Finally, we conducted an evaluation comparing our implementation with the database systems' conventional query optimizer and evaluator.

An analysis of several query plans showed that the quality of hypertree decompositions, i.e., the ability of the implied query plan to reduce the number of redundant tuples quickly, is crucial for the quality of the resulting query plan, and therefore the ability to compete with conventional query evaluation. We therefore added measures to keep the quality of hypertree decompositions high, including the creation of multiple hypertree decompositions competing with each other, considering the physical plan costs of their implied plans.

The results of our evaluation indicate benefits for structure guided query evaluation for some of our test queries, with 40% of the full enumeration queries being executed with lower runtime than with the original DBMS. Furthermore, the need to temporarily store intermediate results for the execution of Yannakakis' algorithm caused the memory usage to be slightly increased. However, we avoid an exponential blowup of intermediate results through Yannakakis' algorithm. Interestingly, a majority (55%) of full enumeration queries whose result is not empty was solved with a lower runtime by our implementation than with conventional query execution. Considering 0MA queries, the original database implementation heavily uses coprocessor tasks, leveraging performance benefits through parallelization across the database cluster, which, in return, causes a worse result for our implementation. Despite these optimizations, structure guided query evaluation still manages to be more performant for about a third of all instances solved by both versions. Finally, our implementation causes 45% less timeouts or out-of-memory errors for 0MA queries than the conventional evaluator.

The results mentioned above clearly show the meaningfulness of structure guided query optimization, deeply integrated in the DBMS, as an additional option in a database systems' query optimizer.

6.1 Future Work

We started to deeply integrate structure guided query optimization into a NewSQL database, namely TiDB. In the course of this thesis, even more possibilities to profit from this technique came up. One interesting follow-up research question would be how the integration of counting algorithms, in order to fully support aggregate operations, and therefore broaden the effects of structure guided query optimization, affects query execution.

Furthermore, an additional step would be the integration of the same techniques into the distributed storage. Consequently, some operations could be loaded off to the storage nodes, benefiting from increased parallelization, and more memory capacity, in case the location of the data across the distributed storage cluster, i.e., data locality, allows to do so. We would expect a major advantage for structure guided query optimization through this measure.

Finally, investigating the field of query plan cost comparison between conventional plans and structure guided query plans, possibly infering the cost from the hypertree decomposition, or directly from created plans, would be a further step towards sensible use of deeply integrated structure guided query execution.

List of Figures

2.1 Example Relation [EN11, Figure 3.1] 8
2.2 Example Foreign Key Relationship 8
2.3 Illustration of an Ordered Database File [EN11, Figure 17.7] 10
2.4 Illustration of a NewSQL Cluster with Database and Storage Nodes . . . 13
2.5 Illustration of the Described Layers 15
2.6 TiDB Architecture of a Basic Cluster [Pin23b] 17
2.7 Multi-Raft System as Part of TiDB [Pin23k] 20
2.8 Balanced Storage Cluster 22

3.1 Cyclic and Acyclic Query 28
3.2 Join Tree of the Query Q_{CQ} 29
3.3 Join Tree of the Query Q_{JT_Ex} 30
3.4 Query Hypergraph of the Query Q_{HG_Ex} 31
3.5 (G)HD of the Query Q_{HD_Ex} 34
3.6 Yannakakis Example Part 1 38
3.7 Yannakakis Example Part 2 39

4.1 Current Implementation Architecture of TiDB - SQL Layer [Pin23j] . . . 43
4.2 CTE Usage Example 48
4.3 Plan Builder with Structure Guided Query Evaluation 49
4.4 Example of a Simplified Hypertree Decomposition 53
4.5 Resulting Join Trees After Bottom-Up Semi-Join 53
4.6 Resulting Join Trees After Top-Down Semi-Join 55
4.7 Resulting Join Trees After Bottom-Up Join 55
4.8 Example for Building an Expression Between Two HD Nodes 57
4.9 Plan Builder with Structure Guided Query Evaluation Including JoinCosts 61
4.10 New Implementation Architecture of TiDB - SQL Layer 62

5.1 Test Flow 65
5.2 CTEMock Evaluation: Memory Difference Plots 67
5.3 CTEMock Evaluation: Runtime Scatterplots 68
5.4 Join Costs Evaluation: Cartesian Plans and Full Runtimes 69
5.5 Early Stop Evaluation: Runtime Scatterplots 70
5.6 Secondary Plans & Join Costs Evaluation: Runtime over 5 Runs 73

5.7 Final Evaluation: Full Enumeration Overall Runtime Scatterplot 75
5.8 Final Evaluation: Full Enumeration Runtime Differences 75
5.9 Final Evaluation: Full Enumeration Memory Usage Differences 76
5.10 Final Evaluation: Full Enumeration Zero Result Queries Runtimes Scatterplot 77
5.11 Final Evaluation: Full Enumeration Non-Zero Result Queries Runtimes . 77
5.12 Final Evaluation: 0MA Overall Runtime Scatterplot 78
5.13 Execution Plan of 04aa as 0MA Query with the Original Version 81
5.14 Comparing two plans: 05al by the Original Version vs. Our Implementation 82

List of Tables

2.1 Most Common Normal Forms [EN11, Chapter 15.3 onwards] 9

5.1 Test Cluster Setup 64
5.2 Secondary Plans & Join Costs Evaluation: Improvements (%) over Init Plan 72
5.3 Secondary Plans & Join Costs Evaluation: Additional Planning Time (ms) 72
5.4 Final Evaluation: Full Enumeration Groups and Results 76
5.5 Final Evaluation: 0MA Groups and Results 79

List of Code Listings

4.1 RecordSet Interface 44
4.2 Executor Interface 46
4.3 Pseudocode for Creating a Hypergraph 50
4.4 Pseudocode for First Bottom-Up Traversal of Plan Builder 52
4.5 Pseudocode for Top-Down Traversal of Plan Builder 54
4.6 Pseudocode for Bottom-Up Join Traversal of Plan Builder 56

Acronyms

0MA	Zero-Materialization Answerable. 5, 6, 40, 49, 51, 57, 65–67, 74, 77–79, 81, 83, 84
2PC	Two-Phase Commit. 16, 17
AST	Abstract Syntax Tree. 22, 43, 44, 49
CRUD	Create/Read/Update/Delete. 13
CTE	Common Table Expression. 45, 46, 48, 51–59, 67, 74, 80, 81, 83
DBMS	Database Management System. 1–7, 10, 12–16, 25, 80, 83, 84
DDL	Data Definition Language. 10
DML	Data Manipulation Language. 10
DQL	Data Query Language. 10, 26, 43
ETL	Extract/Transform/Load Operation. 14
HTAP	Hybrid Transaction/Analytical Processing. 14, 15
JSON	Javascript Object Notation. 12
MVCC	Multi Version Concurrency Control. 17, 18
NoSQL	Not Only SQL. 1, 7, 12–14
OLAP	Online Analytical Processing. 14, 16
OLTP	Online Transaction Processing. 14, 16
RDBMS	Relational Database Management System. 6–14, 17, 26
SQL	Structured Query Language. 1, 7, 10, 11, 13–16, 22, 25, 26, 31, 43, 44, 50

Bibliography

[Aba12] Daniel Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42, February 2012. Conference Name: Computer.

[AJR+17] Foto N. Afrati, Manas R. Joglekar, Christopher M. Re, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A Multiround Distributed Join Algorithm. In Michael Benedikt and Giorgio Orsi, editors, *20th International Conference on Database Theory (ICDT 2017)*, volume 68 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:18, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 1868-8969.

[ALOR18] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Re. LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 449–460, April 2018. ISSN: 2375-026X.

[ALT+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Transactions on Database Systems*, 42(4):20:1–20:44, October 2017.

[BB16] Johann Brault-Baron. Hypergraph Acyclicity Revisited. *ACM Computing Surveys*, 49(3):54:1–54:26, December 2016.

[Cat11] Rick Cattell. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, 39(4):12–27, May 2011.

[CDD+23] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. Differentially Testing Database Transactions for Fun and Profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, pages 1–12, New York, NY, USA, January 2023. Association for Computing Machinery.

[CHMA21] Ionela Chereja, Sarah Myriam Lydia Hahn, Oliviu Matei, and Anca Avram. Multidimensional Analysis of NewSQL Database Systems. In Radek Silhavy, editor, *Software Engineering and Algorithms*, Lecture Notes in Networks and Systems, pages 221–236, Cham, 2021. Springer International Publishing.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 77–90, New York, NY, USA, May 1977. Association for Computing Machinery.

[Coc23] CockroachLabs. cockroachdb/cockroach, June 2023. `https://github.com/cockroachdb/cockroach`, accessed: 2023-06-16.

[EN11] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Addison-Wesley, Boston, 6th ed edition, 2011. OCLC: ocn586123196.

[Fag83] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, July 1983.

[FGLP21] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. *ACM Journal of Experimental Algorithmics*, 26:1.6:1–1.6:40, July 2021.

[FGP18] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and Fractional Hypertree Decompositions: Hard and Easy Cases. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '18, pages 17–32, New York, NY, USA, May 2018. Association for Computing Machinery.

[GGGS07] Lucantonio Ghionna, Luigi Granata, Gianluigi Greco, and Francesco Scarcello. Hypertree Decompositions for Query Optimization. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 36–45, April 2007. ISSN: 2375-026X.

[GGLS16] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 57–74, New York, NY, USA, June 2016. Association for Computing Machinery.

[GGS11] Lucantonio Ghionna, Gianluigi Greco, and Francesco Scarcello. H-DB: a hybrid quantitative-structural sql optimizer. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 2573–2576, New York, NY, USA, October 2011. Association for Computing Machinery.

[GHTC13] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22, December 2013.

[GLL$^+$23] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice, March 2023. arXiv:2303.02723 [cs], Including New Version May 2023.

[GLOP22] Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler. Fast Parallel Hypertree Decompositions in Logarithmic Recursion Depth. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '22, pages 325–336, New York, NY, USA, June 2022. Association for Computing Machinery.

[GLS99] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On Tractable Queries and Constraints. In Trevor J.M. Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *Database and Expert Systems Applications*, Lecture Notes in Computer Science, pages 1–15, Berlin, Heidelberg, 1999. Springer.

[GLS01] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 195–206, New York, NY, USA, May 2001. Association for Computing Machinery.

[GLS02] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, May 2002.

[GM93] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, April 1993.

[GOP22] Georg Gottlob, Cem Okulmus, and Reinhard Pichler. Fast and parallel decomposition of constraint satisfaction problems. *Constraints*, 27(3):284–326, July 2022.

[Gra79] Mark H. Graham. *On the universal relation*. University of Toronto, 1979.

[Gra94] Goetz Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, February 1994. Conference Name: IEEE Transactions on Knowledge and Data Engineering.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[GS09] Georg Gottlob and Marko Samer. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics*, 13:1:1.1–1:1.19, February 2009.

[HLC+20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, August 2020.

[ISO] ISO/IEC. ANSI SQL 1999: sql1999.pdf. `https://web.cecs.pdx.edu/~len/sql1999.pdf`, accessed: 2023-06-10.

[IUV17] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1259–1274, New York, NY, USA, May 2017. Association for Computing Machinery.

[Lam19] Leslie Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. Association for Computing Machinery, New York, NY, USA, October 2019.

[MF23] MetaBrainz Foundation. MusicBrainz – The open Music-Encyclopedia, October 2023. `https://musicbrainz.org/`, accessed: 2023-10-19, Current Data on: `http://ftp.musicbrainz.org/pub/musicbrainz/data/`.

[MKC+22] Riccardo Mancini, Srinivas Karthik, Bikash Chandra, Vasilis Mageirakos, and Anastasia Ailamaki. Efficient massively parallel join optimization for large queries. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 122–135, New York, NY, USA, 2022. Association for Computing Machinery.

[Mon] MongoDB, Inc. ACID Properties In DBMS Explained | MongoDB | MongoDB. `https://www.mongodb.com/basics/acid-transactions`, accessed: 2023-06-07.

[NR18] 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, pages 677–692, New York, NY, USA, May 2018. Association for Computing Machinery.

[OO14] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[OV20] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer International Publishing, Cham, 2020.

[Pin23a] PingCap. GitHub - pingcap/tiflow: This repo maintains DM (a data migration platform) and TiCDC (change data capture for TiDB), June 2023. https://github.com/pingcap/tiflow, accessed: 2023-06-08.

[Pin23b] PingCap. TiDB - Github, June 2023. https://github.com/pingcap/tidb, accessed: 2023-06-12.

[Pin23c] PingCap. TiDB Architecture, June 2023. https://docs.pingcap.com/tidb/stable/tidb-architecture, accessed: 2023-06-15.

[Pin23d] PingCap. TiDB Computing, June 2023. https://docs.pingcap.com/tidb/stable/tidb-computing, accessed: 2023-06-16.

[Pin23e] PingCap. TiDB Dev Guide, October 2023. https://pingcap.github.io/tidb-dev-guide/index.html, accessed: 2023-10-05.

[Pin23f] PingCap. TiDB Dev Guide: DQL, October 2023. https://pingcap.github.io/tidb-dev-guide/understand-tidb/dql.html, accessed: 2023-10-05.

[Pin23g] PingCap. TiDB Optimizer Hints, October 2023. https://docs.pingcap.com/tidb/stable/optimizer-hints, accessed: 2023-10-19.

[Pin23h] PingCap. TiDB Scheduling, June 2023. https://docs.pingcap.com/tidb/stable/tidb-scheduling, accessed: 2023-06-15.

[Pin23i] PingCap. TiDB Software And Hardware Recommendations, October 2023. https://docs.pingcap.com/tidb/dev/hardware-and-software-requirements#development-and-test-environments, accessed: 2023-10-19.

[Pin23j] PingCap. TiDB SQL Optimization Concepts, October 2023. https://docs.pingcap.com/tidb/dev/sql-optimization-concepts, accessed: 2023-10-05.

[Pin23k] PingCap. TiDB Storage, June 2023. https://docs.pingcap.com/tidb/stable/tidb-storage, accessed: 2023-06-16.

[Pin23l] PingCap. TiDB System Variables - Memory Quota, October 2023. https://docs.pingcap.com/tidb/stable/system-variables#tidb_mem_quota_query, accessed: 2023-10-19.

[Pin23m] PingCap. TiDB Transaction Isolation Levels, June 2023. https://docs.pingcap.com/tidb/stable/transaction-isolation-levels#tidb-transaction-isolation-levels, accessed: 2023-06-13.

[Pin23n] PingCap. TiSpark, June 2023. https://github.com/pingcap/tispark, accessed: 2023-06-08.

[Pin23o] PingCap. TiUP Overview, June 2023. `https://docs.pingcap.com/tidb/stable/tiup-overview`, accessed: 2023-06-08.

[PS13] Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 79(6):984–1001, September 2013.

[RG11] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill higher education. McGraw-Hill, Boston, 3rd ed., intern. ed. 2003, 14. [print.] edition, 2011.

[SAC+79] Patricia Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas 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, pages 23–34, New York, NY, USA, May 1979. Association for Computing Machinery.

[SGL04] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, pages 210–221, New York, NY, USA, June 2004. Association for Computing Machinery.

[TiK23] TiKV Authors. PD, June 2023. `https://github.com/tikv/pd`, accessed: 2023-06-15.

[TR15] Susan Tu and Christopher Ré. DunceCap: Query Plans Using Generalized Hypertree Decompositions. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 2077–2078, New York, NY, USA, May 2015. Association for Computing Machinery.

[VG84] Patrick Valduriez and Georges Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Transactions on Database Systems*, 9(1):133–161, March 1984.

[VJPO21] Patrick Valduriez, Ricardo Jimenez-Peris, and M. Tamer Özsu. Distributed Database Systems: The Case for NewSQL. In Abdelkader Hameurlain and A. Min Tjoa, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems XLVIII: Special Issue In Memory of Univ. Prof. Dr. Roland Wagner*, Lecture Notes in Computer Science, pages 1–15. Springer, Berlin, Heidelberg, 2021.

[WY21] Yilei Wang and Ke Yi. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 1969–1981, New York, NY, USA, June 2021. Association for Computing Machinery.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 82–94, Cannes, France, September 1981. VLDB Endowment.

[YO79] Clement Tak Yu and Meral Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, pages 306–312, November 1979.