TU WIEN Informatics

Lightweight Integration of Query Decomposition Techniques into SQL-based Database Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Alexander Selzer, BSc
Matrikelnummer 01633655

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Reinhard Pichler
Mitwirkung: Dipl.-Ing. Matthias Lanzinger
Dipl.-Ing. Davide Longo
Dipl.-Ing. Cem Okulmus

Wien, 8. Februar 2021

Alexander Selzer — Reinhard Pichler

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

Lightweight Integration of Query Decomposition Techniques into SQL-based Database Systems

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Alexander Selzer, BSc
Registration Number 01633655

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Reinhard Pichler
Assistance: Dipl.-Ing. Matthias Lanzinger
Dipl.-Ing. Davide Longo
Dipl.-Ing. Cem Okulmus

Vienna, 8th February, 2021

Alexander Selzer Reinhard Pichler

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Alexander Selzer, 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, 8. Februar 2021

Alexander Selzer

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Die Berechnung der Ergebnisse von Join-Queries ist eine der zentralen Herausforderungen in jedem Datenbanksystem und hat direkte Auswirkungen auf die Geschwindigkeit sowie den Ressourcenverbrauch. Das Problem, SQL-Queries zu beantworten, ist äquivalent zu dem *Conjunctive Queries* (CQs) zu beantworten, welches ein NP-vollständiges Problem ist. Daher ist der Aufwand eines Joins über mehrere Tabellen in einer Datenbank exponentiell in der Anzahl der Tabellen im worst-case. Die Suche nach effizient lösbaren Fällen des CQ Problems war Gegenstand intensiver Forschung über mehrere Jahrzehnte. Eins der größeren Ergebnisse der Forschung in diesem Bereich ist das Resultat, dass *azyklische Conjunctive Queries* (ACQs) effizient lösbar sind. Angesichts der Tatsache, dass viele reale Queries zyklisch sind, sind ACQs eine starke Einschränkung. Die *Hypertree-Width* ist eine Verallgemeinerung der Azyklizität, welche die Klasse der azyklischen Queries in eine Hierarchie von *fast-azyklischen* Queries, welche in polynomieller Zeit lösbar sind, ausweitet. Die Laufzeit der Query-Ausführung ist allerdings von einer guten *Hypertree-Zerlegung* abhängig. Algorithmen, um diese effizient zu berechnen, sind kürzlich erschienen, was den Lösungsansatz realistisch macht.

Obwohl dieser Ansatz zur Query-Ausführung vielversprechend ist, wurde er bis jetzt in kein bestehendes relationales DBMS integriert. Nur ein Prototyp ist bekannt, von dem jedoch die Implementierungsdetails nicht verfügbar sind. In dieser Arbeit wird ein System zur Struktureller-Zerlegung-basierten Query-Optimierung entwickelt. Da eine Integration in den Kern des DBMS komplex wäre und die Wiederverwendung von Zerlegungsalgorithmen schwer machen würde, wurde eine Integration außerhalb des DBMS entwickelt. Solch ein System kann verwendet werden, ohne den DBMS Server zu ersetzen, und könnte mehrere DBMS Implementierungen unterstützen. Um die Laufzeit noch weiter zu verbessern, werden Statistiken aus der Datenbank extrahiert und verwendet, um bessere Zerlegungen zu finden. Zusätzlich wird die Query-Ausführung parallelisiert.

Der Vergleich zeigt, dass unser System mit PostgreSQL mithalten kann. Obwohl es auf den meisten Instanzen langsamer oder vergleichbar schnell ist, identifizieren wir Fälle in denen es das DBMS übertrifft. Bei *boolean CQs*, welche beantworten, ob Ergebnisse existieren oder nicht, stellte sich unser System als sehr effektiv heraus und erzielte wesentlich bessere Ergebnisse als das DBMS alleine. Die Verwendung von Statistiken erwies sich als essenziell für gute Laufzeiten. Des weiteren zeigen wir, dass die dadurch ermöglichte parallele Ausführung von Queries die Performance verbessern kann.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

The evaluation of join queries is a central challenge in every database system, which directly affects the query-answering speed and resource utilization. Answering *conjunctive queries* (CQs), a problem equivalent to that of answering SQL *select-from-where* statements, is an NP-complete problem. Consequently, joining multiple tables in a database requires an exponential effort with respect to the number of tables, in the worst case. A substantial amount of research has therefore been dedicated to the search for tractable fragments of the CQ evaluation problem. One of the major outcomes of research in this field is the result that *acyclic conjunctive queries* (ACQs) can be answered efficiently. Considering that many real-world queries are cyclic, ACQs are a strong restriction. The notion of *hypertree-width* provides a generalization of acyclicity, widening the class of acyclic queries into a hierarchy of *nearly-acyclic* queries solvable in polynomial time. The runtime of the query execution is, however, dependent on good *hypertree decompositions*. Algorithms to compute these quickly have emerged recently, making the approach feasible.

Although this approach towards query execution is promising, it has not yet been integrated into any existing relational DBMS. Only one research prototype is known where the implementation details are unavailable. In this thesis, a system for structural decomposition-based query optimization is developed and evaluated. Since an integration into the core of the DBMS is complex and makes the reuse of decomposition tools difficult, we have implemented a lightweight integration outside of the DBMS. Such a lightweight system can be integrated by users without replacing the DBMS server and may support different DBMS implementations. For further performance improvements, statistical information is extracted from the database and used for the computation of a good decomposition, and a parallel execution strategy is implemented.

The experimental evaluation showed that our system is competitive with PostgreSQL on the CQ answer enumeration problem. Although performing worse or comparably on most instances, we identified cases where it outperforms the DBMS significantly. On the problem of evaluating *boolean CQs*, queries determining whether a matching row exists or not, the system proved to be very effective and outperformed the DBMS on many instances. We also conclude that the integration of statistics into the computation of the decomposition is essential for the competitiveness of the system. Furthermore, the parallel execution strategy is shown to provide a performance improvement on multiple instances.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**

2 Structural Decompositions **5**
2.1 Hypergraphs and Decompositions 5
2.2 Further Generalizations of (Generalized) Hypertree-Width 12
2.3 Hypergraph Invariants 12
2.4 Features of Hypertree Decompositions 13

3 Computing Hypertree Decompositions **17**
3.1 opt-k-decomp 18
3.2 k-decomp 21
3.3 det-k-decomp 23
3.4 (new-)det-k-decomp for GHDs 25
3.5 BalancedGo 27
3.6 Generating GHDs by covering Tree Decompositions 28
3.7 Further Approaches 29
3.8 Hypergraph Benchmarks 29

4 Query Processing **31**
4.1 Conjunctive Queries 31
4.2 Acyclic Conjunctive Queries 33
4.3 Yannakakis' Algorithm 36
4.4 Query Optimization 40
4.5 Join Algorithms 42
4.6 Query Optimization in PostgreSQL 44

5 Integrating Structural Decompositions into Database Systems **51**
5.1 State of the Art 51

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.2 Overview of our System . . . 53
5.3 Weighted Hypertree Decompositions . . . 54
5.4 The Query Optimization/Execution Pipeline . . . 57
5.5 Optimized Query Execution . . . 62
5.6 Hypergraph and Hypertree Formats . . . 65
5.7 Hypergraph Visualization . . . 69

6 Experimental Evaluation **73**
6.1 Benchmarking . . . 73
6.2 Overview of Results . . . 78

7 Conclusion **81**
7.1 Future Work . . . 81

A Benchmark Results **83**
A.1 Optimized vs. Non-Optimized Performance . . . 84
A.2 Performance on Boolean Queries . . . 88
A.3 The Effect of applying Weighted Decompositions . . . 92
A.4 Parallel Execution vs. PL/pgSQL Execution . . . 96
A.5 Parallel vs. Sequential Execution . . . 99

B Benchmarking Data **103**
B.1 db1 . . . 103
B.2 db2 . . . 105
B.3 db3 . . . 109
B.4 db4 . . . 114
B.5 db5 . . . 118
B.6 db6 . . . 119
B.7 fat_and_cycles . . . 120
B.8 tpc-h . . . 121
B.9 tpc-h-2 . . . 123

Bibliography **127**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

The evaluation of SQL *select-from-where* queries, a class equivalent to *conjunctive queries*, and arguably the most important class of queries due to their widespread applications in mainstream systems. It is, however, an NP-complete problem [Chandra and Merlin, 1977]. Evaluating a join over multiple tables, as frequently necessary in real-world systems, requires exponential effort (with respect to the size of the query) in the worst case. Unsurprisingly, given the possible impact of such findings, several decades of research have been devoted towards the identification of tractable subsets of conjunctive queries. One of the major outcomes of research in this field was the result by Yannakakis that acyclic conjunctive queries can be evaluated in linear time [Yannakakis, 1981]. Strictly acyclic queries are tractable due to the fact that a join tree with one table per node can be found, and the query can hence evaluated in linear time via a sequence of semi joins.

Acyclicity is a strong restriction, and queries are in practice not strictly acyclic but still close to acyclicity in most cases. Therefore, the generalization of acyclicity goes beyond this restricting notion, relaxing the criteria, and widening the class of acyclic queries into a hierarchy of *nearly-acyclic* queries, solvable in polynomial time [Gottlob et al., 2002, 2016]. The degree of acyclicity of a query is given by the hypertree-width, a property of the hypergraph-structure of the query over the database, corresponding to the width of the minimal-width hypertree decomposition. To evaluate queries efficiently, we use the algorithm of Yannakakis, which depends on a minimal-width hypertree decomposition in order to run efficiently and works by executing semi-joins along the decomposition tree. The worst-case runtime is then bounded by the largest set in the decomposition, which has to be fully joined before Yannakakis' algorithm can be applied, resulting in a runtime only polynomial in the size of this set. Still, finding a minimal-width hypertree decomposition is a major challenge. A program for parallelized computation has recently been developed by Gottlob et al. (2020c) and can solve most practical instances in milliseconds.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

While Yannakakis' algorithm and hypertree decompositions can be used to evaluate SQL join queries of bounded hypertree-width in polynomial time, existing DBMS do not make any use of these techniques. Instead, they apply the same decade-old approach of exploring the search space of join orderings evaluated using statistical estimations [Leis et al., 2015] or heuristic search algorithms. While there has been one integration of Yannakakis' algorithm in combination with statistical information into an existing system, the code and implementation details are no longer available. Nevertheless, this shows that there is interest in the topic [Ghionna et al., 2007]. The novel approach towards query optimization utilizing structural decompositions has also been integrated into at least two advanced research prototypes, showing very promising results [Aberger et al., 2017; Aref et al., 2015]. As of the time of writing, no attempt to achieve a lightweight integration into an existing DBMS, i.e. without changing the internals, is known. Structural query optimization methods could have the potential to improve the state of the art of query processing. While the integration of these techniques would have been unrealistic in the past, recently, technologies to generate hypertree decompositions efficiently have matured, making an integration into existing systems feasible [Gottlob et al., 2020a,c].

An implementation in the core of the DBMS would, due to the complexity of the query engine, be a challenging task and would make the reuse of existing programs for generating decompositions difficult. Therefore, a lightweight integration on top of the existing system is of great interest as a first step towards full integration into existing database systems. Furthermore, such a system has the advantage that it can be used by users to optimize the query execution even if they have installed a standard version of a DBMS. Thus, the goal of this work is to produce such a lightweight integration of a decomposition-based optimization approach into an existing system, as well as providing a comprehensive overview of the current state of structural decomposition based query evaluation.

As part of the thesis, a software package implementing a query optimization pipeline was developed in the form of a Java library. The pipeline (using the parallel execution strategy and statistics) consists of the following steps:

1. Analyze the SQL query structure with respect to the database schema and generate a hypergraph
2. Extract statistics from the database and build a weighted hypergraph
3. Decompose the weighted hypergraph into a hypertree decomposition through the use of existing decomposition tools
4. Convert the hypertree decomposition to a join tree, removing redundant attributes
5. Generate a parallel execution ordering of SQL statements implementing Yannakakis' algorithm over the join tree
6. Execute the SQL statements in parallel

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Two approaches were implemented for query execution. One is based on PL/PgSQL functions while the second is a novel approach to parallelize query execution. To compare the performance of the rewritten query with the baseline query, and the effect of different optimization parameters, a benchmarking tool is also developed.

In the first chapter, we introduce hypergraphs and the notion of structural decompositions. The second chapter covers the computation of structural decompositions, from the very first algorithms to the state-of-the-art techniques. The main prerequisites, conjunctive queries, the acyclicity of queries, and query optimization in database systems, are described in chapter 3. In chapter 4, we review the state of the art of structural decomposition-based optimizations and explain the implementation details and design choices of our system. The results of the experimental evaluation of the system are presented and discussed in the 5. chapter. Detailed descriptions of the data used and experimental results can be found in the appendices.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Structural Decompositions

Tree decompositions, hypertree decompositions, generalized hypertree decompositions, as well as the further generalization of fractional hypertree decompositions, are structural decomposition techniques for breaking up structures and problems into smaller parts. Hypertree decompositions are of great interest due to their applications in query optimization. We will first introduce tree decompositions. Then we extend the concept to hypertree decompositions and further to generalized hypertree decompositions, as well as briefly cover further generalizations. Finally, we will consider how hypergraphs can be characterised via several invariants, and features extracted from hypertree decompositions.

2.1 Hypergraphs and Decompositions

Graphs with binary edges can be used to effectively represent many problems. Yet, in some situations, they are not the best representation, and a generalization is of advantage. To achieve a representation with better semantics and to retain the information about relations between vertices directly as edges, hypergraphs can be applied. Hypergraphs are a generalization of binary graphs permitting edges with more than two vertices.

Definition 2.1.1 (Hypergraph)**.** A hypergraph $\mathcal{H} = (V, E)$ consists of a set of vertices V and a set of hyperedges $E \subseteq 2^V \setminus \{\emptyset\}$.

Trees are a particularly simple class of graphs. Many NP-hard problems in computer science become solvable in polynomial time when restricted to tree inputs. For example, consider the well-known MINIMUM-VERTEX-COVER problem of finding, for a graph $G = (V, E)$, a minimal set of vertices $S \subseteq V$ such that for each edge $e \in E$, at least one of the vertices in S is incident. MINIMUM-VERTEX-COVER is NP-complete and only solvable in exponential time in the general case. However, if the input graph is a tree, or in other words, a connected undirected acyclic graph, the problem suddenly becomes easily

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

solvable. Due to the fact that, in a tree, no cycles can occur, we know that, for every tree node, everything "below" this node is unconnected to anything above it. We can thus apply a simple dynamic programming algorithm to solve the Minimum-Vertex-Cover problem in polynomial time ($O(n^2)$) [Valiente, 2013].

In search of larger classes of tractable fragments, a natural generalization was made from full acyclicity (trees) to "almost" acyclic hypergraphs (we will cover the acyclicity of hypergraphs in more detail in chapter 4). Acyclicity can be quantified as the quality of fit (*width*) of the best way to decompose the hypergraph into a tree and can thus be seen as the similarity of the hypergraph to a tree [Robertson and Seymour, 1986].

Definition 2.1.2 (Tree Decomposition)**.** A ***tree decomposition*** of a hypergraph $G = (V, E)$ is a pair $(\mathcal{T}, \chi)$ where $\mathcal{T} = (T, F)$ is a tree, $\chi : T \to 2^V$ is a labeling function assigning to each node $p \in T$ a set of vertices $\chi(p) \subseteq V$, and the following conditions are satisfied:

(1) For each vertex $v \in V$, there exists a tree node $t \in T$ such that $v \in \chi(t)$

(2) For each edge $e \in E$, there exists a tree node $t \in T$ such that $e \subseteq \chi(t)$

(3) For each vertex $v \in V$, $\{t \in T \mid v \in \chi(p)\}$ induces a connected subtree

[Gottlob et al., 2014; Flum and Grohe, 2006]

If the relevant class of graphs has no isolated vertices, condition (1) can be dropped, since in this case, condition (2) is sufficient for ensuring that every vertex will occur in the tree decomposition.

As seen in Definition 2.1.2, a tree decomposition is a labeled tree where each tree node has an associated set of vertices from the decomposed graph. Condition (1) ensures that all of the graph's vertices are covered by the tree and condition (2) ensures that each pair of vertices connected by an edge occurs together in one of the tree nodes. Condition (3) (known as the *connectedness-condition*) guarantees that all occurrences of a vertex in the tree are connected. This last condition is essential for the dynamic programming approaches on tree decompositions to work by allowing all partial solutions to be combined (joined).

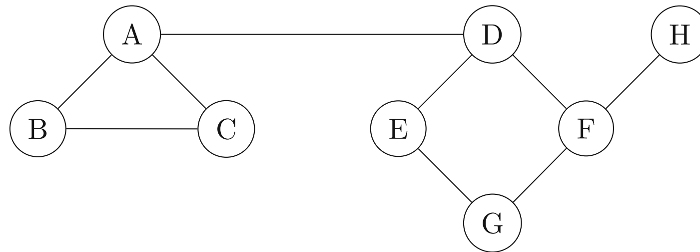


Figure 2.1: An 8-vertex graph with some degree of cyclicity

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

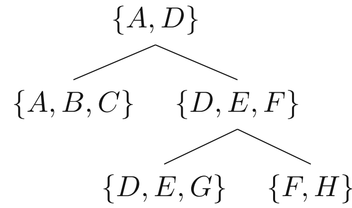


Figure 2.2: A tree decomposition of the graph in Figure 2.1

In Figure 2.1 we see a graph with two clear cycles. Vertices $A - B - C$ form a triangle, and vertices $D - E - F - G$ form a cycle of length 4. The decomposition in Figure 2.2 is one of several possible tree decompositions.

Definition 2.1.3 (Tree-Width)**.** The ***width*** over a tree decomposition $\mathcal{T} = (T, F)$ of a graph $G = (V, E)$ is the size of its largest tree-node:

$$max_{v \in V}(\chi(v)) - 1 \tag{2.1}$$

The ***tree-width*** of a graph $G = (V, E)$, denoted as $tw(G)$, is the minimum width over all possible tree decompositions of G.

In Figure 2.2, the width of our tree decomposition is 2, and it is minimal. One example of a non-optimal tree decomposition of width 4 is shown in Figure 2.3, where the larger tree-node containing $\{D, E, F, G, H\}$ could be broken up further by pulling out H which is only connected by one edge to the cycle.

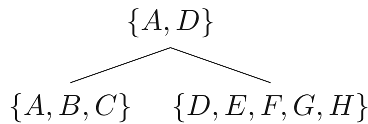


Figure 2.3: A width 4 tree decomposition of the graph in Figure 2.1

Interestingly, the standard definition of tree-width of a graph is not equal to the size of the largest tree-node, but 1 less. The reason behind this choice is the fact that any graph with at least one edge will have a set of at least size 2 in its decomposition. Hence, per the definition of tree-width, the class of trees has a tree-width of 1. Decomposing a tree is straight-forward, since its structure can be directly re-used in the tree decomposition.

A classic use-case of hypergraphs is the representation of relationships between variables in *constraint satisfaction problems (CSP)*. CSPs consist of a set of variables $X = \{X_1, ..., X_n\}$, a set of domains corresponding to the legal assignments of the variables $D = \{D_1, ..., D_n\}$, and a set of constraints restricting which values the variables can take and defining the relationships between the variables [Russell and Norvig, 2002].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Example 2.1.1. Consider a CSP with variables $X = \{A, B, C, D, E\}$, domain $\{1, ..., 100\}$ for each variable and the following constraints:

$$C_1 : A + B + C + D = 60 \tag{2.2}$$

$$C_2 : C \cdot D \cdot E = 40 \tag{2.3}$$

$$C_3 : A \neq E \tag{2.4}$$

The CSP in Example 2.1.1 contains three constraints of different arity: C_1 constrains 4 variables, C_2 constrains 3 variables and C_3 is a binary constraint. A direct representation of which variables constrain each other in form of a hypergraph is seen in Figure 2.4.

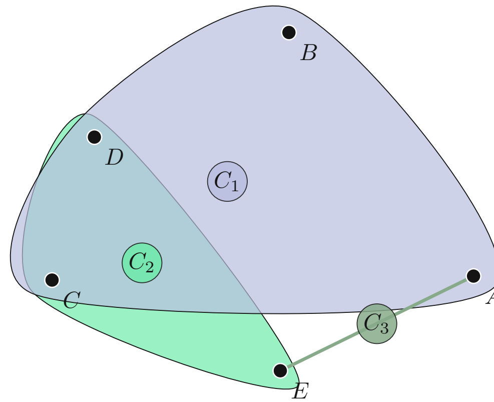


Figure 2.4: A hypergraph representation of the problem in Example 2.1.1

Among the successful use cases of hypertree decomposition are CSPs, as well as the core focus of this thesis - *join query evaluation* - which we will cover in chapter 4.

Definition 2.1.4 (Hypertree)**.** A ***hypertree*** associated with a hypergraph $\mathcal{H}$ is a triple (T, χ, λ) where $T = (N, E)$ is a rooted tree and $\chi : T \rightarrow 2^{vertices(\mathcal{H})}$ and $\lambda : T \rightarrow 2^{edges(\mathcal{H})}$ are labeling functions which for each tree node $p \in N$, assign a set of vertices $\chi(p) \subseteq vertices(\mathcal{H})$ and a set of edges $\lambda(p) \subseteq edges(\mathcal{H})$.

From now on, we refer by *node* to a tree node $p \in N$ and by *vertex* to a vertex of the hypergraph $v \in V$. The *bag* associated with a node n refers to the set of hyperedges $\lambda(n)$.

We also introduce some syntax to simplify further definitions: For a subtree $T' = (N', E')$, $\chi(T') = \bigcup_{v \in N'} \chi(v)$, i.e. all vertices contained in all bags of the subtree.

Furthermore, for any tree node $p \in N$, T_p denotes the subtree rooted at p [Gottlob et al., 2001b]. We only need to consider a rooted tree T in the case of hypertree decompositions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In generalized hypertree decompositions, the tree can be arbitrarily rooted [Gottlob et al., 2020a].

Definition 2.1.5 (Hypertree Decomposition)**.** A ***hypertree decomposition (HD)*** of a hypergraph $\mathcal{H}$ is a hypertree (T, χ, λ) with $T = (N, E)$ satisfying the following conditions:

(1) for each edge $h \in edges(\mathcal{H})$, there exists a hypertree node $p \in N$ such that $h \subseteq \chi(p)$

(2) for each vertex $v \in vertices(\mathcal{H})$, the set $\{p \in N | v \in \chi(p)\}$ induces a connected subtree of T

(3) for each hypertree node $p \in N$, $\chi(p) \subseteq \left(\bigcup_{e \in \lambda(p)} e\right)$

(4) for each hypertree node $p \in N$, $\left(\bigcup_{e \in \lambda(p)} e\right) \cap \chi(T_p) \subseteq \chi(p)$

The ***width*** of a hypertree decomposition (T, χ, λ) with $T = (N, E)$ is the largest number of hypergraph edges associated with a tree node $max_{v \in N} |\lambda(v)|$ and the ***hypertree-width*** $hw(\mathcal{H})$ of a hypergraph refers to the smallest possible width over all hypertree decompositions.

Naturally, the definition of hypertree decompositions is similar to that of tree decompositions, especially conditions 1-3. Condition (1) of Definition 2.1.5 ensures that all edges of the hypergraph are covered by the hypertree, condition (2) ensures the connectedness of subtrees induced by hypergraph vertices, and condition (3) guarantees that all vertices of the hypertree node are contained in the edges of the node. Condition (4) is a special condition (referred to as the *descendant condition*), which was introduced to simplify the computation of hypertree decompositions by Gottlob et al. (2002). It makes sure that, for each hypertree node $p \in N$, if a hypergraph vertex exists in the hyperedge set $\bigcup_{e \in \lambda(p)} e$ as well as in the vertex sets $\chi(T_p)$ of the subtree rooted at the node p, it also has to exist in the vertex set $\chi(p)$ [Gottlob et al., 2016].

Generalized hypertree decompositions are a simpler and more intuitive variant resulting in the dropping of condition (4) from Definition 2.1.5.

Definition 2.1.6 (Generalized Hypertree Decomposition)**.** A ***generalized hypertree decomposition (GHD)*** of a hypergraph $\mathcal{H}$ is a hypertree (T, χ, λ) with $T = (N, E)$ satisfying the following conditions:

(1) for each edge $h \in edges(\mathcal{H})$, there exists a hypertree node $p \in N$ such that $h \subseteq \chi(p)$

(2) for each vertex $v \in vertices(\mathcal{H})$, the set $\{p \in N \mid v \in \chi(p)\}$ induces a connected subtree of T

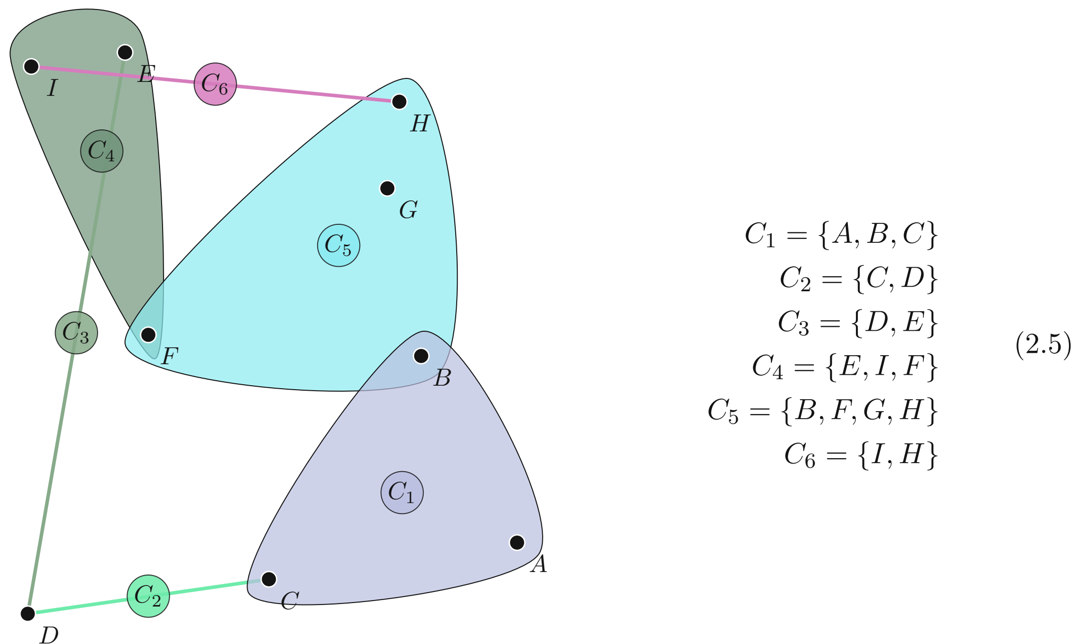
(3) for each hypertree node $p \in N$, $\chi(p) \subseteq \left(\bigcup_{e \in \lambda(p)} e\right)$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The *width* of GHDs and the *generalized-hypertree-width* of a hypergraph $ghw(\mathcal{H})$ are defined analogously to 2.1.5.

The generalization from hypertree decompositions to generalized hypertree decompositions comes at the cost of tractability. Finding a hypertree decomposition for a fixed width was shown to be tractable by Gottlob et al. (2002), whereas finding a generalized hypertree decomposition of width k for $k \geq 3$ was shown to be NP-hard. Since the class of GHDs is broader than that of HDs, for a hypertree, a GHD of lower width than the best-width HD may exist [Gottlob et al., 2016]. However, for a hypergraph or a class of hypergraphs, the generalized hypertree-width is not far off from the hypertree-width, and the following relationship holds: $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq 3 \cdot ghw(\mathcal{H}) + 1$. Thus, if the hypertree-width on a class of hypergraphs is bounded, the generalized hypertree-width is bounded as well and vice versa [Adler et al., 2007].

We will now consider decompositions of the slightly cyclic hypergraph in Figure 2.5, which in its structure is not too far off from some typical hypergraphs that might occur in moderately-complex database queries.



$$
\begin{aligned}
C_1 &= \{A, B, C\} \\
C_2 &= \{C, D\} \\
C_3 &= \{D, E\} \\
C_4 &= \{E, I, F\} \\
C_5 &= \{B, F, G, H\} \\
C_6 &= \{I, H\}
\end{aligned}
\tag{2.5}
$$

Figure 2.5: A cyclic hypergraph H_1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

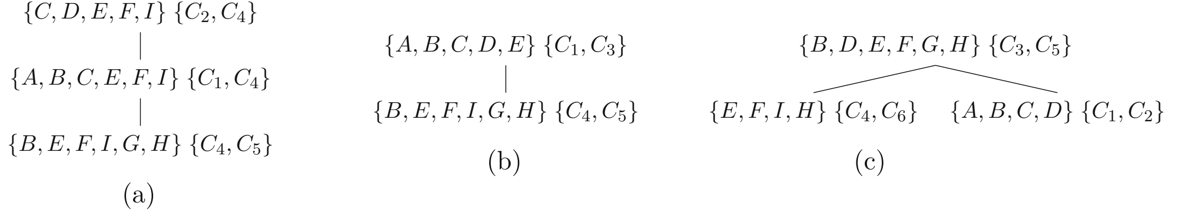


Figure 2.6: GHDs of H_1 (Figure 2.5) (generated using BalancedGo)

In Figure 2.6, we see 3 simple as well as differently-structured decompositions of the same width. From the existence of a width 2 decomposition and the fact that there are cycles in the hypergraph (a width of 1 is impossible), we can conclude that $ghw(H_1) = 2$. In fact, all of the decompositions are not only GHDs but also HDs, since the descendant condition is not violated. Hence, we can conclude $hw(H_1) = 2$.

To illustrate how a GHD may violate the descendant condition, we modify the GHD (a) of Figure 2.6. By removing the vertices $\{E, F, I\}$ from the first tree-node, the hypergraph is still a valid GHD since none of the conditions are violated. However, in this tree-node the vertices $\{E, F, I\}$ are now only part of the edge C_4 of the hyperedge set, but no longer in the vertex set, which is not a violation of the GHD conditions since condition (1) is still fulfilled by the other two tree-nodes, the induced subtrees of condition (2) are not broken and as required by condition (3), the set of vertices is still covered by the set of hyperedges. However, assuming p_1 is the first tree-node (from the top) and p_2 is the second tree-node. Now, $\{E, F, I\} \subsetneq chi(p_1)$ and $\{E, F, I\} \in \lambda(p_1)$ while $\{E, F, I\} \subseteq \chi(p_2)$ and hence $\left(\bigcup_{e \in \lambda(p)} e\right) \cap \chi(T_p) \subsetneq \left(\bigcup_{e \in \lambda(p)} e\right)$.

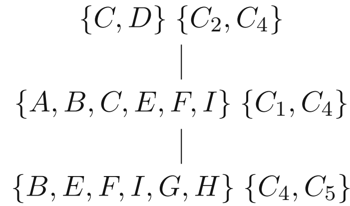


Figure 2.7: A hypertree violating the descendant condition

While hypertree decompositions could be considered as tree decompositions with an additional set of edges per node, it is quickly seen that the tree-width is frequently far off from the hypertree-width. We know that $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq tw(\mathcal{H}) + 1$, but also that classes of graphs with bounded hypertree-width and unbounded tree-width exist [Adler, 2004].

Hypergraph invariants can be seen as *equivalent* if they are within a constant factor of each other. Two hypergraph invariants I and J are equivalent if, for constant factors c, d, $c \cdot I(\mathcal{H}) \leq J(\mathcal{H}) \leq d \cdot I(\mathcal{H})$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Adler et al. (2007) showed that the invariants hypertree-width, generalized hypertree-width, hyperbranch-width, and *minimum number of marshals* with a monotone as well as non-monotone winning strategy are all equivalent, as well as several further equivalence and inequivalence results on other properties such as the *hyperlinkedness* or the *hyperbramble-number*. Therefore, hw and ghw are considered equivalent to each other in this sense but not as equivalent to tree-width.

2.2 Further Generalizations of (Generalized) Hypertree-Width

A further common generalization of GHDs and ghw are *fractional hypertree decompositions (FHDs)* and the corresponding width-measure *fractional hypertree-width (fhw)* [Grohe and Marx, 2006]. The problem of checking for a hypergraph whether $fhw(\mathcal{H}) \leq k$ for $k \geq 2$ was proven NP-complete by Fischl et al. (2018). Since FHDs are a generalization of GHDs, the following relationship holds between the measures of width: $fhw(\mathcal{H}) \leq ghw(\mathcal{H}) \leq hw(\mathcal{H})$.

FHDs can be defined by replacing one of the conditions of GHDs by a more general one and redefining the width-measure. Consider a hypergraph $\mathcal{H} = (V, E)$ and an *edge weight function* $\gamma : V \rightarrow [0, 1]$. The coverage of vertices is given by $B(\gamma) = \{v \in V \mid \sum_{e \in E, v \in E} \gamma(e) \geq 1\}$ and the weight of the function by $weight(\gamma) = \sum_{e \in E} \gamma(e)$. γ is a *fractional edge cover* of a subset of the vertices $X \subseteq V$ when $X \subseteq B(\gamma)$.

By replacing λ with γ_p or each tree-node p and replacing condition (3) of Definition 2.1.6 with the following, we gain the definition of an FHD:

(3) for each hypertree node $p \in N$, $\chi(p) \subseteq B(\gamma_p)$

The width of an FHD is defined as the maximum width $width(\gamma_p)$ over all tree nodes and the fractional hypertree-width $fhw(\mathcal{H})$ is given by the smallest width over all FHDs of $\mathcal{H}$.

For a set $X \subseteq V$, $\rho^*_H(X)$ denotes the minimum weight over all fractional edge covers of X. In the case of an *integral edge cover*, the edge weight functions are restricted to integral values such that they are defined as $\gamma : E \rightarrow \{0, 1\}$. $\rho_H(X)$ denotes the minimum weight over all integral edge covers of X.

2.3 Hypergraph Invariants

The decision problems Check(HD, k), Check(GHD, k) and Check(FHD, k) are defined as the problems of checking whether an HD, GHD, or FHD of width at most k exists. Due to the NP-completeness of Check(GHD, k) and Check(FHD, k) [Fischl et al., 2018, 2019; Gottlob et al., 2020b], efforts were made to identify tractable fragments

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of hypergraphs. There exist several favourable properties of hypergraphs, which allow grouping them into easy-to-solve classes. We will go into more detail about the problems of computing decompositions in Section 3.

Definition 2.3.1 (bounded degree)**.** Analogously to binary graphs, the degree $deg(H)$ is defined as the maximum number of hyperedges in which a vertex occurs.

A class $\mathcal{C}$ of hypergraphs has a bounded degree (BDP) if there exists a $d \geq 1$ such that every hypergraph $H \in \mathcal{C}$ has $deg(H) \leq d$.

Definition 2.3.2 (BIP)**.** The $iwidth(H)$ *(intersection width)* of a hypergraph is the cardinality of the largest intersection $|e_1 \cap e_2|$ of two hyperedges $e_1 \neq e_2$.

A hypergraph satisfies the *i-bounded intersection property (i-BIP)* if $iwidth(H) \leq i$.

A class of hypergraphs $\mathcal{C}$ has the BIP if there exists an i such that all hypergraphs $H \in \mathcal{C}$ have the *i-BIP*.

Definition 2.3.3 (BMIP)**.** The *c-multi-intersection width* $c-miwidth(H)$, for a positive integer c, is the cardinality of the largest intersection $|e_1 \cap \cdots \cap e_c|$ of c distinct edges $e_1, ..., e_c$.

A hypergraph satisfies the *i-bounded c-multi-intersection property (ic-BMIP)* if $c - miwidth(H) \leq i$. The *ic-BMIP* with $c = 2$ is the *i-BIP*.

A class of hypergraphs $\mathcal{C}$ has the *bounded multi-intersection property (BMIP)* if there exist constants c and i such that every graph $H \in \mathcal{C}$ has the ic-BMIP.

Hypergraphs of d-bounded c-multi-intersection property with $d \geq 0$ and $c \geq 1$ are also referred to as (c, d)-hypergraphs [Gottlob et al., 2020a].

Definition 2.3.4 (VC Dimension)**.** Consider a hypergraph $H = (V, E)$ and a subset of the vertices $X \subseteq V$. The *trace* of E on X is defined as $E|_X = \{X \cap e | e \in E\}$. X is said to be *shattered* if $E|_X = 2^X$.

The *Vapnik-Chervonenkis dimension (VC-dimension)* is the maximum cardinality of a shattered subset of V.

A class $\mathcal{C}$ of hypergraphs has *bounded VC-dimension* if there exists a $v \geq 1$ such that every hypergraph $H \in \mathcal{C}$ has a VC-dimension $\leq v$

The class of hypergraphs of bounded degree is contained in the class of BMIP hypergraphs, which is in turn contained in the class of bounded VC-dimension hypergraphs. These properties can be seen as a measure of hypergraph-complexity.

2.4 Features of Hypertree Decompositions

The computational complexity of dynamic programming algorithms on tree decompositions or (G)HDs is usually polynomial to the degree of the width of the decomposition or

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

(ghw). In practical implementations of these algorithms, however, the concrete structure of the decomposition can lead to a significant difference in the algorithm's runtime. Multiple decompositions of the same width may thus have very different runtimes. Since the computational effort of finding the decompositions is in general small compared to the effort of running the actual dynamic programming algorithm, and heuristics can easily generate structurally very different decompositions, the approach of generating multiple minimal-width decompositions and choosing the best is a promising approach at reducing the total runtime. Abseher et al. (2017) successfully applied this approach to MSO-solvers. *MSO (Monadic Second-Order Logic)* is an extension of first-order logic in which problems taking structures of bounded tree-width as input are tractable. *MSO-solvers* take as input an MSO-formula as well as a structure and solve a decision or optimization problem. The two solvers D-FLAT and SEQUOIA work with dynamic programming on tree decompositions. Many common decision or optimization problems, such as 3-Colorability and Minimum-Dominating-Set, can be defined in MSO and are thus tractable on bounded-tree-width inputs. The implementation generates various distinct tree decompositions and measures quantifiable features of them as well as the runtime of the MSO solvers mentioned above. Decompositions are then ranked by relative performance. A regression model is trained to predict the rank from the decomposition features.

In order to train the regression model, Abseher et al. (2017) identified an exhaustive list of features ranging from simple properties such as the average bag size to more sophisticated measures such as the bag connectedness ratio [Michael Abseher and Woltran, 2017]. Although these features are formally defined on tree decompositions, one can consider hypertree decompositions as tree decompositions by leaving out the sets of hyperedges.

Here we present a list of features of (generalized) hypertree decompositions. Consider a GHD (T, χ, λ) with $T = (N, E)$ over a hypergraph $\mathcal{H} = (V, C)$. Some features are direct numeric measures (e.g. *NodeCount*) of the hypertree while the other features are sets of measures that can be aggregated (e.g. *VertexBagSize*) using statistic functions such as the mean or the minimum.

NodeCount

$$NodeCount = |N| \tag{2.6}$$

Depth The depth of the tree

VertexBagSize

$$VertexBagSize = \left\{ p \in N \;\middle|\; |\chi(p)| \right\} \tag{2.7}$$

Statistics: *mean, median, min, max, stdev, sum*

EdgeBagSize

$$EdgeBagSize = \left\{ p \in N \;\middle|\; |\lambda(p)| \right\} \tag{2.8}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Statistics: *mean, median, min, max, stdev, sum*

Note: $max(EdgeBagSize)$ is the width of the decomposition

VertexContainerCount

$$VertexContainerCount = \left\{ v \in V \;\middle|\; |\{p \in N | v \in \chi(p)\}| \right\} \tag{2.9}$$

Statistics: *mean, median, min, max, stdev, sum*

EdgeContainerCount

$$EdgeContainerCount = \left\{ e \in C \;\middle|\; |\{p \in N | e \in \lambda(p)\}| \right\} \tag{2.10}$$

Statistics: *mean, median, min, max, stdev, sum*

VertexItemLifetime The heights of the subtrees induced by each hypergraph vertex

$$VertexItemLifetime = \left\{ v \in V \;\middle|\; height(T[\{p \in N | v \in \chi(p)\}]) \right\} \tag{2.11}$$

where $T[S]$ with $S \subseteq N$ denotes the induced subtree of the nodes S on T

Statistics: *mean, median, min, max, stdev*

EdgeItemLifetime The heights of the subtrees induced by each hyperedge

$$EdgeItemLifetime = \left\{ e \in C \;\middle|\; height(T[\{p \in N | e \in \lambda(p)\}]) \right\} \tag{2.12}$$

Statistics: *mean, median, min, max, stdev*

EdgeItemLifetime The heights of the subtrees induced by each hyperedge

$$EdgeItemLifetime = \left\{ e \in C \;\middle|\; height(T[\{p \in N | e \in \lambda(p)\}]) \right\} \tag{2.13}$$

Statistics: *mean, median, min, max, stdev*

LeafNodeCount The number of leaf nodes

JoinNodeCount The number of join nodes (tree nodes with more than one successor)

LeafNodePercentage The percentage of leaf nodes in the tree

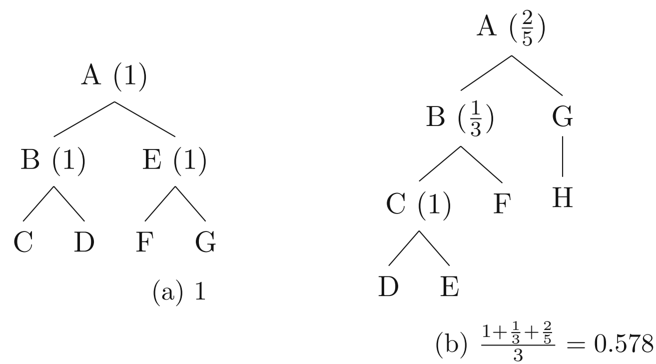
JoinNodePercentage The percentage of join nodes in the tree

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

BalancednessFactor A measure of balancedness of the decomposition tree.

The sizes of subtrees are summed up and when a join node is reached, the size of the smaller tree is divided by the size of the larger tree. Finally, the mean of all join nodes is calculated. A value closer to 0 indicates an unbalanced tree while a value closer to 1 indicates a balanced tree.

As an example, consider the computation of the balancedness of the trees in Figure 2.8



(a) 1

(b) $\frac{1+\frac{1}{3}+\frac{2}{5}}{3} = 0.578$

Figure 2.8: Two trees and their balancedness factor

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 3

Computing Hypertree Decompositions

In this chapter, we go over the complexity of computing decompositions and decomposition algorithms, starting from the early approaches *opt-k-decomp* and *k-decomp* up to the current state-of-the-art parallel computation techniques.

Although the computation of CQ problems becomes tractable if a GHD is known, due to the issue that finding an optimal or a bounded-width GHD is NP-complete, the two-step process of computing a GHD and then evaluating the query is also NP-complete. HDs improve on this, making the process tractable if the hypertree-width is bounded, however at the cost of greater width and hence a larger polynomial runtime.

There exist two general approaches towards generating (G)HDs: *(1) computing tree decompositions and covering their vertex sets* and *(2) hypergraph-specific algorithms*.

Since, in practice, algorithms for finding optimal decompositions can become quite expensive, *(1) optimal algorithms* (e.g. *det-k-decomp*), *(2) heuristic algorithms* (e.g. genetic algorithms), and *hybrid approaches* (e.g. *det-k-decomp* with heuristics) are used.

Before covering hypertree decomposition algorithms, we define the notions of separators and balanced separators as they are used in the following decomposition algorithms.

Definition 3.0.1 (Separators)**.** Consider a hypergraph $H = (V, E)$ and a subset of the vertices $S \subseteq V$. A set $C \subseteq (V \setminus S)$ is *[S]-connected* if for any two distinct vertices $v, w \in C$, there exists a sequence of vertices $v = v_0, ..., v_h = w$ and a sequence of edges $e_0, ..., e_{h-1}$ with $h \geq 0$ such that $\{v_i, v_{i+1}\} \subseteq (e_i \setminus S)$ for each $i \in \{0, ..., h-1\}$. In other words, any two vertices in C are connected by a path along the hyperedges, which is not blocked by the vertices of S.

A set $C \subseteq V$ is an *[S]-component* if C is [S]-connected and maximal. Such a set S is referred to as a *separator* and gives rise to disjoint subsets $E_C = \{e \in E \mid e \cap C \neq \emptyset\}$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The *size* of an $[S]$-component is defined as $|E_C|$. For a hypertree (T, χ, λ) and a tree node $v \in V(T)$, a $[v]$*-component* is defined as a $[\chi(v)]$-component.

S is a *balanced separator* if all $[S]$-components of H have size $\leq \frac{|E|}{2}$ [Gottlob et al., 2020a].

Definition 3.0.2 (k-vertex)**.** A *k-vertex* $R \subseteq E(\mathcal{H})$ refers to a set of hyperedges with $|R| \leq k$. A $[R]$-component of a k-vertex R is defined as a $[\bigcup_{e \in R} e]$-component. k-vertices are later used as hyperedge sets in λ, thus the restriction to $\leq k$ was made.

3.1 opt-k-decomp

Introduced by Gottlob et al. (1999), *opt-k-decomp* was the first polynomial-time algorithm for computing optimal HDs. *opt-k-decomp* takes a hypergraph $\mathcal{H}$ and an integer parameter k. If an HD of $hw(\mathcal{H}) \leq k$ exists, such a decomposition is returned; otherwise, the algorithm terminates without a result, in which case we know the hypergraph has a hypertree-width greater than k. While it is possible to find more decompositions by increasing k, choosing a low k is of advantage since the runtime of the algorithm is polynomial to a degree of $2k$.

For the purposes of opt-k-decomp, a restriction to HDs in *normal form* was made. Optimality with respect to all possible HDs is not lost, since or every HD of width k of a hypergraph, there exists a normal form HD of width k.

Definition 3.1.1 (H)**.** A hypertree decomposition (T, χ, λ) with $T = (N, E)$ of a hypergraph $\mathcal{H}$ is in *normal form* if, for each tree node $r \in N$ and each child node s of n, the following conditions hold:

(1) There is exactly one $[r]$-component C_r such that $\chi(T_s) = C_r \cup (\chi(s) \cap \chi(r))$

(2) $\chi(s) \cap C_r \neq \emptyset$

(3) $vertices(\lambda(s)) \cap \chi(r) \subseteq \chi(s)$

opt-k-decomp runs in 3 major steps: 1. constructing a partitioned directed graph (Algorithm 1) 2. computing the weights of the graph (Algorithm 2) 3. constructing the final decomposition (Algorithm 3) The pseudo-code was taken from the original paper by Gottlob et al. (1999) and slightly modified for consistency with the rest of the thesis.

In the first step, the directed graph $CG := (N_e \cup N_a, A, weight)$ is built up. The set N_e consists of entries of the form (R, C) where R is a k-vertex and C is a $[R]$-component. The node $(root, V) \in N_e$ is a special node without any outgoing arcs representing the root of the decomposition. Elements $(R, C) \in N_e$ represent sub-hypergraphs and the elements $(S, C) \in N_a$ with an arc to (R, C) potential candidates for breaking up the sub-hypergraph. Furthermore, all elements $(S, C') \in N_a$ with $C' \subseteq C$ are required for the computation and connected by arcs to (S, C).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 1 ComputeCG

```
procedure ComputeCG(H = (V, E), k)
    CG := (N_e ∪ N_a, A, weight) with weight : N_e ∪ N_a → ℕ
    N_e := {(root, V)} ∪ {(R, C)|R is a k-vertex and C is a [R] − component
    N_a := ∅; A := ∅
    for all (R, C) ∈ N_e do
        rc := (⋃_{h∈E(C)} vertices(h)) ∩ vertices(R)
        for all k-vertex S do
            if vertices(S) ∩ C ≠ ∅ ∧ rc ⊆ vertices(S) then
                N_a := N_a ∪ {(S, C)}
                Add an arc from (S, C) to (R, C) in A
                for all (S, C′) ∈ N_e with C′ ⊆ C do
                    Add an arc from (S, C′) to (S, C) in A
                end for
            end if
        end for
    end for
    return CG
end procedure
```

In the next step, the weights of the arcs $(u, v) \in A$ are computed. All vertices with no incoming arcs are initialized to ∞. Then, all vertices $(S, C) \in N_e$ are assigned the minimum weight of all the adjacent edges in N_a, and all vertices in $(S, C) \in N_a$ are assigned what is greater of the following: the weight of one of the vertices in N_a with an incoming node or the size of the k-vertex $|S|$. After performing the weighting procedure, the weight of the special root node $weight((root, V(\mathcal{H})))$ is equal to the hypertree-width $hw(\mathcal{H})$ in the case that $hw(\mathcal{H}) \leq k$, otherwise $weight((root, V(\mathcal{H}))) = \infty$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 2 WeightCG

```
procedure WeightCG(H = (V, E), CG := (N_e ∪ N_a, A, weight))
    for all (R, C) ∈ N_e with no incoming arcs in CG do
        weight((R, C)) := ∞
    end for
    for all (S, C) ∈ N_a with no incoming arcs in CG do
        weight((S, C)) := ∞
    end for
    while there is an unweighted vertex p = (S, C) in N_e ∪ N_a do
        if p ∈ N_e then
            weight(p) = min({weight(q) | (q, p) ∈ A})
        else
            weight(p) = max({|S|} ∪ {weight(q) | (q, p) ∈ A})
        end if
    end while
end procedure
```

Finally, if a decomposition of width $\leq k$ exists, it is constructed from the directed graph. By starting from the root node $(root, V(\mathcal{H}))$ and recursively choosing the lowest-weighted vertices, an HD of minimal width in normal form is constructed.

The total runtime of opt-k-decomp is in $O(|E|^{2k} \cdot |V|^2)$ and follows from the size of the graph there being $O(|E|^{2k} \cdot |V|)$ vertices in N_a, where each vertex has $O(|V|)$ incoming arcs. In practice, however, although the asymptotic complexity might not seem as dramatic, the algorithm fails due to the large amounts of memory required by the bottom-up approach of computing the directed graph and its weights [Gottlob and Samer, 2008]. Computing and storing all k-vertices, of which there are $O(|E|^k)$ becomes the major driver of computational cost on large instances. A later improved version was developed by Harvey and Ghose (2003), which managed to improve the runtime of opt-k-decomp significantly by restricting the normal form of HDs even further and thus getting closer to the best-case complexity for most instances. However, despite the promising improvements, this version of opt-k-decomp breaks down on large instances as well.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 3 ComputeHypertree

```
procedure ComputeHypertree(p, r)
    Choose a minimum-vertex-weighted predecessor (S, C) of p
    Create a new vertex s as a child of r in T
    λ(s) := S
    χ(s) := vertices(S) ∩ (C ∪ χ(r))
    for all predecessor q of (S, C) do
        ComputeHypertree(q,s)
    end for
end procedure
procedure OPT-K-DECOMP(H = (V, E), k)
    CG := ComputeCG(H)
    WeightCG(CG)
    if weight(root, V) = ∞ then
        return failure
    else
        HD = (T, χ, λ)
        Create a vertex root′ in T
        χ(root′) := ∅
        λ(root′) := ∅
        return ComputeHypertree((root, V), root')
    end if
end procedure
```

3.2 k-decomp

Introduced by Gottlob et al. (2002), *k-decomp* was a different approach to the computation of HDs with a focus on showing complexity results. k-decomp is formulated as a non-deterministic algorithm running on an *alternating Turing machine (ATM)*. By the statement of this ATM-algorithm, Gottlob et al. were able to prove that the problem of finding an HD of width $\leq k$ is in LogCFL - a favourable result, since LogCFL is a class of highly-parallelizable problems. As we will see soon, while k-decomp is mainly of theoretical interest, *det-k-decomp* is based on k-decomp, and works by replacing the non-deterministic part of the algorithm with a search procedure.

The class LogCFL consists of all problems logspace reducible to a context-free language. It stands in the following relation to some other well-known complexity classes:

$$\mathrm{AC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{LogCFL} \subseteq \mathrm{AC}^1 \subseteq \mathrm{NC}^2 \subseteq P \tag{3.1}$$

Due to the containment in NC^2, problems in LogCFL are solvable in logarithmic time by a CRCW (concurrent-read-concurrent-write) PRAM with a polynomial number of processors. Therefore, by scaling up the number of processors polynomially, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

polynomial complexity of the problem can be offset. An ATM is a non-deterministic Turing machine defined similarly to a standard *deterministic Turing machine*, where all states are either *existential states* or *universal states* [Chandra and Stockmeyer, 1976]. An existential state accepts if at least one of the outgoing transitions accepts while a universal state accepts if all outgoing transitions end up accepting. An algorithm defined as an ATM consists of a tree of alternating existential and universal states, the size of which is the *computation tree size*. The class LogCFL can also be characterized by decision problems definable on an ATM running with computation tree size $O(n^{O(1)})$ and with space usage in $O(\log n)$.

Algorithm 4 k-decomp

1: **procedure** K-DECOMPOSABLE($Component, OldSep, \mathcal{H} = (V, E)$)
2:     **guess** a separator $S \subseteq E$ s.t. $|S| \leq k$
3:     **check** that the following conditions hold:
4:         $vertices(Component) \cap vertices(OldSep) \subseteq vertices(S)$
5:         $S \cap Component \neq \emptyset$
6:     if one of the checks fails return null
7:     $\mathcal{C} := \{C \in E \mid C$ is an [S]-component and $C \subseteq Component\}$
8:     $Subtrees := \emptyset$
9:     **for all** $C \in \mathcal{C}$ **do**
10:         $HD :=$ k-decomposable(C, S)
11:         **if** $HD = null$ **then**
12:             return null
13:         **else**
14:             $Subtrees := Subtrees \cup \{HD\}$
15:         **end if**
16:     **end for**
17:     $\chi = (vertices(Component) \cap vertices(OldSep)) \cup vertices(S \cap Component)$
18:     return $createHDNode(S, \chi, Subtrees)$
19: **end procedure**
20: **procedure** K-DECOMP($\mathcal{H} = (V, E)$)
21:     return k-decomposable$(E, \emptyset, \mathcal{H})$
22: **end procedure**

Algorithm 4 contains the procedure for k-decomp, which is an adaptation by Gottlob and Samer (2008) of the original algorithm from Gottlob et al. (2002). k-decomp consists of a recursive procedure *k-decomposable*, which takes as parameters two sets of hyperedges: a separator *OldSep* and an [OldSep]-component *Edges*. Lines 2-5 of *k-decomposable* are the non-deterministic guess-and-check part, which guesses a separator $S \subseteq E$ and then checks the conditions necessary for a valid HD: connectedness to the parent-separator (line 4) and being part of the correct component i.e., sub-hypergraph (line 5). Then, all components created by the separator are enumerated, and the procedure *k-decomp*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is recursively called. At the end, all sub-hypertrees are appended to the newly created hypertree node. Gottlob et al. considered a possible implementation of the algorithm on an ATM and showed that the non-deterministic part of the algorithm can run with polynomial computation tree size and the data-structures which may be used take up only logarithmic space. Therefore, k-decomp is in LogCFL.

3.3 det-k-decomp

det-k-decomp, by Gottlob and Samer (2008), is an implementable version of k-decomp which replaces the non-deterministic guess-and-check part by a heuristic search procedure. Through the use of heuristics for speeding up the search, it is not guaranteed that an optimal HD will be found, but if $hw(\mathcal{H}) \leq k$, an HD of width $\leq k$ will be found. A further normal form, the *strong normal form*, is defined, and the search is restricted to decompositions of this normal form.

The main procedure of algorithm 5 calls the procedure *decompCov*, which calls *decompAdd*, and which in turn calls *decompSub*, which finally again calls *decompCov*, making the last 3 effectively one recursive procedure, that was split up for better readability [Gottlob and Samer, 2008]. The sets *FailedSeps* and *SuccSeps* are considered global variables, and are used to store the already visited successfully and unsuccessfully computed separators in order to avoid exponential runtime by visiting them again.

Algorithm 5 det-k-decomp

1: **procedure** DET-K-DECOMP($\mathcal{H} = (V, E)$)
2: $\quad FailSeps := \emptyset$
3: $\quad SuccSeps := \emptyset$
4: $\quad HD = decompCov(E, \emptyset, \mathcal{H})$
5: $\quad$ **if** $HD \neq$ null **then**
6: $\quad\quad HD := expand(HD)$
7: $\quad$ **end if**
8: $\quad$ return HD
9: **end procedure**

Procedure *decompCov* computes the λ-labels of the hypergraph. First, it checks whether the component is of size $\leq k$ and could be trivially decomposed. Next, for each potential separator fulfilling the condition on line 4 of Algorithm 4 computed by *cover* using ordering heuristics for better runtime, *decompAdd* is called. *decompAdd* then checks whether the condition on line 5 of Algorithm 4 can be satisfied by the separator, backtracking if not. If the condition is already satisfied, no edge is added; otherwise, an edge is added to the separator to fulfill the condition. For all subcomponents created by separating the component, *decompSub* is called, which again calls *decompCov* and makes sure that no already computed separators are computed again.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 6 decompCov

```
 1: procedure DECOMPCOV(Edges ⊆ E, Conn ⊆ V, H = (V, E))
 2:     if |Edges| ≤ k then
 3:         HD := createHDNode(Edges, vertices(Edges, ∅))
 4:         return HD
 5:     end if
 6:     BoundEdges := {e ∈ E | e ∩ Conn ≠ ∅}
 7:     for all CovSep ∈ cover(Conn, BoundEdges) do
 8:         HD := decompAdd(Edges, Conn, CovSep, H)
 9:         if HD ≠ null then
10:             return HD
11:         end if
12:     end for
13:     return null
14: end procedure
```

Algorithm 7 decompAdd

```
 1: procedure DECOMPADD(Edges ⊆ E, Conn ⊆ V, CovSep, H = (V, E))
 2:     InCovSep := CovSep ∩ Edges
 3:     if InCovSep ≠ ∅ ∨ k − |CovSep| > 0 then
 4:         if InCovSep = ∅ then
 5:             AddSize := 1
 6:         else
 7:             AddSize := 0
 8:         end if
 9:         for all AddSep ⊆ Edges with |AddSep| = AddSize do
10:             Separator := CovSep ∪ AddSep
11:             Components := separate(Edges, Separator)
12:             if ∀Comp ∈ Components : (Separator, Comp) ∉ FailSeps then
13:                 Subtrees := decompSub(Components, Separator, H)
14:                 if Subtrees ≠ ∅ then
15:                     χ := Conn ∪ vertices(InCovSep ∪ AddSep)
16:                     HD = createHDNode(Separator, χ, Subtrees)
17:                     return HD
18:                 end if
19:             end if
20:         end for
21:     end if
22:     return null
23: end procedure
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 8 decompSub

```
 1: procedure DECOMPSUB(Components, Separator, H = (V, E))
 2:     Subtrees := ∅
 3:     for all Comp ∈ Components do
 4:         ChildConn := vertices(Comp) ∩ vertices(Separator)
 5:         if (Separator, Comp) ∈ SuccSeps then
 6:             HD := createHDNode(Comp, ChildConn, ∅)
 7:         else
 8:             HD := decompCov(Comp, ChildConn, H)
 9:             if HD = null then
10:                 FailSeps := FailSeps ∪ {(Separator, Comp)}
11:                 return ∅
12:             else
13:                 SuccSeps := SuccSeps ∪ {(Separator, Comp)}
14:             end if
15:         end if
16:         Subtrees = Subtrees ∪ {HD}
17:     end for
18:     return Subtrees
19: end procedure
```

det-k-decomp runs in time $O(|E|^k \cdot min(|E| \cdot d^k, |E|^k) \cdot min(|V|, |E|)^2)$ where d is the maximum number of incident hyperedges over all hyperedges and in space $O(|E|^k + min(|E|, |V|) \cdot (|E| + |V|))$. While det-k-decomp runs in polynomial time, as opt-k-decomp, on practical instances, the empirical evaluation of the algorithm by Gottlob and Samer (2008) showed the memory usage to be significantly lower than that of opt-k-decomp. For example, on a hypergraph of 139 vertices and 133 hyperedges, opt-k-decomp quickly uses up the RAM and is prevented from performing actual computation while det-k-decomp, in comparison, allocates a minuscule amount of memory. Gottlob et al. performed benchmarks of det-k-decomp and opt-k-decomp on several datasets, such as *DaimlerCrysler* and *Graph2D* (see Section 3.8). det-k-decomp significantly outperforms opt-k-decomp, which in most cases does not terminate in the given time. It also outperforms the heuristic bucket elimination approach by Dermaku et al. (2008), which achieves decompositions of worse width.

3.4 (new-)det-k-decomp for GHDs

Fischl et al. (2018) identified tractable classes of GHDs and FHDs, which allow solving the problems CHECK(GHD, k) and CHECK(FHD, k) in polynomial time. These tractable subsets are based on the hypergraph properties from Section 2.3. Such favourable classes turned out to cover a large part of the existing benchmark instances (see Section 3.8).

CHECK(GHD, k) becomes tractable for a fixed $k \geq 1$ on the class of hypergraphs enjoying

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

the BIP. The *GHD det-k-decomp* algorithm introduced by Fischl et al. is parameterized not only by k, but also by the intersection width d, allowing it to detect $(2, d)$-hypergraphs of $ghw(\mathcal{H}) \leq k$. It effectively reduces the problem of computing a GHD of bounded intersection width to the problem of computing an HD. The majority of hypergraphs found in the HyperBench instances based on real-world problems are $(2, d)$-hypergraphs of low d (less than 3); therefore, the runtime of these instances can be bounded to a low polynomial. Unfortunately, the *GHD det-k-decomp* algorithm turned out not to be effective in practice, but the authors also introduced a new approach based on balanced separators. This new approach turned out to be very effective, reducing the number of instances with unknown ghw noticeably and forms the basis for BalancedGo.

As part of *new-det-k-decomp*, the `GlobalBIP` and `LocalBIP` algorithms were implemented. They solve Check(GHD) in polynomial time on hypergraphs having the BIP [Fischl et al., 2019].

`GlobalBIP` extends the hypergraph's hyperedges by $f(\mathcal{H}, k)$, which contains, for each hyperedge $e \in E(\mathcal{H})$, all intersections of e with up to k other hyperedges.

$$f(\mathcal{H}, k) = \bigcup_{e \in E(\mathcal{H})} \left(\bigcup_{e_1, \ldots, e_j \in (E(\mathcal{H}) \setminus \{e\}), j \leq k} 2^{e \cap (e_1 \cup \ldots \cup e_j)} \right) \tag{3.2}$$

The size of $f(\mathcal{H}, k)$ is bounded by k and the intersection size d, containing at most $d \cdot k$ elements. Det-k-decomp (or new-det-k-decomp for HDs) can then be called on the extended hypergraph $\mathcal{H}' = (V(\mathcal{H}), E(\mathcal{H}) \cup f(\mathcal{H}k))$, and will run in polynomial time since the number of hyperedges stays polynomial. If it terminates and returns an HD, this HD can be converted back to a GHD of the original hypergraph efficiently.

Algorithm 9 The `GlobalBIP` algorithm

1: **procedure** GLOBALBIP($\mathcal{H} = (V, E), k \geq 1$)
2: compute $f(\mathcal{H}, k)$
3: $H' := (V, E \cup f(\mathcal{H}, k))$
4: $D := NewDetKDecomp(H', k)$
5: **if** $H \neq NULL$ **then**
6: **for all** $u \in D$ **do**
7: **for all** $e \in (\lambda(u) \cap f(\mathcal{H}, k))$ **do**
8: $e' := e' \in E$ such that $e \subseteq e'$
9: $\lambda(u) = (\lambda(u) \setminus \{e\}) \cup e'$
10: **end for**
11: **end for**
12: **end if**
13: return D
14: **end procedure**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

LocalBIP is based on the same approach as GlobalBIP, but reduces the amount of extra hyperedges needed by computing these "locally" with respect to the hypertree bags, thus the new-det-k-decomp procedure is modified.

The new-det-k-decomp software was open-sourced and made available as an easy-to-use commandline utility written in C++.[1] It was also integrated into the system developed in this thesis.

3.5 BalancedGo

Based on the balanced separator approach by Fischl et al. (2019), a highly effective parallel implementation with further improvements was developed by Gottlob et al. (2020c). Written in Go, the software makes use of the well-known built-in concurrency features of the language [Donovan and Kernighan, 2015]. Its source code is also open-source and publicly available.[2]

The main idea behind the balanced separator approach is the fact that every GHD must contain a node u whose edge set $\lambda(u)$ is a balanced separator, since the root can be arbitrarily chosen in such a way that the left subcomponent is smaller than half the number of hyperedges. By choosing balanced separators as bags, it can be guaranteed that the recursion depth is logarithmically bounded, as opposed to det-k-decomp, where it is linear in the number of edges.

BalancedGo is based on a recursive divide-and-conquer procedure, which first searches in parallel for a balanced separator and then decomposes the resulting components in parallel, thus achieving a high level of parallelism.

Besides the speed-up through parallelization, some algorithmic improvements were made:

- The hypergraph given as input to the algorithm is first preprocessed to make it smaller and simpler to decompose. First, the GYO reduction (see Section 4.2) is applied to simplify the hypergraph. Secondly, all vertices belonging to the same set of edges are reduced to one vertex.
- An improved ordering is applied when searching for balanced separators
- The search space is reduced by avoiding the computation of the same subsets of χ

The authors noticed that the parallel balanced separator approach worked very well at detecting negative instances and splitting larger hypergraphs into subhypergraphs but performed less well on smaller hypergraphs. Therefore, they developed a hybrid approach where the balanced separator algorithm is first used for a fixed number of rounds, then the smaller components are solved with the det-k-decomp algorithm. Due to the ability of

[1] https://github.com/TUfischl/newdetkdecomp
[2] https://github.com/cem-okulmus/BalancedGo

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the *parallel balanced separator* approach to detect negative cases quickly (a lower bound to k) and of the *hybrid* approach to detect positive cases quickly (an upper bound to k), the two were combined. The result was an extremely effective ensemble algorithm which managed to determine the *ghw* of 2320 hyperbench instances, improving greatly on the 1642 instances by new-det-k-decomp.

3.6 Generating GHDs by covering Tree Decompositions

The methods described and applied by Dermaku et al. (2008) make use of the fact that a GHD can be retrieved from a tree decomposition of the hypergraph by covering the vertex sets with hyperedges. Thus, algorithms for generating tree decompositions can be re-used for GHDs. Finding a minimum-width tree decomposition of the hypergraph does not guarantee that the best-covered decomposition is an optimal GHD. However, it still acts as an effective heuristic for a good GHD since a small vertex set χ correlates with a small edge set λ.

Tree decompositions can also be characterized and reconstructed through an *elimination ordering* of a graph. A *triangulated graph*, also referred to as a *chordal graph*, is a graph where each cycle of length greater than or equal to 4 contains a chord, where a chord is an edge connecting two non-adjacent vertices in the cycle. Every triangulated graph has a *perfect elimination ordering*. A perfect elimination ordering $\sigma = (e_1, e_2, ..., e_{|V|})$ of a triangulated graph $G = (V, E)$ is a permutation of the graph's edges, where, for each $i \in \{1, ..., |V|\}$, e_i and its adjacent vertices form a clique in the induced subgraph $G[\sigma(i), ..., \sigma(|V|)]$. Given a perfect elimination ordering, a triangulated graph can be fully reconstructed. In the case of an arbitrary non-triangulated graph, the definition is weakened to that of an elimination ordering. In an elimination ordering, any permutation of vertices is valid, however, the graph cannot be reconstructed directly, but rather a triangulated version of it. For more detail, refer to Musliu (2008)

Any permutation leads to a valid tree decomposition; thus, the challenge lies in finding a good permutation, leading to low tree-width. Many algorithms for computing tree decompositions make use of this and represent solutions by elimination orderings. This problem representation is especially well-suited for applying heuristic algorithms, such as simulated annealing, genetic algorithms [Musliu, 2008], or tabu search, due to its simplicity (a fixed-length list of vertices) and the fact that any permutation is a valid solution.

As an example, consider the graph from Figure 3.1 and its tree decomposition.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

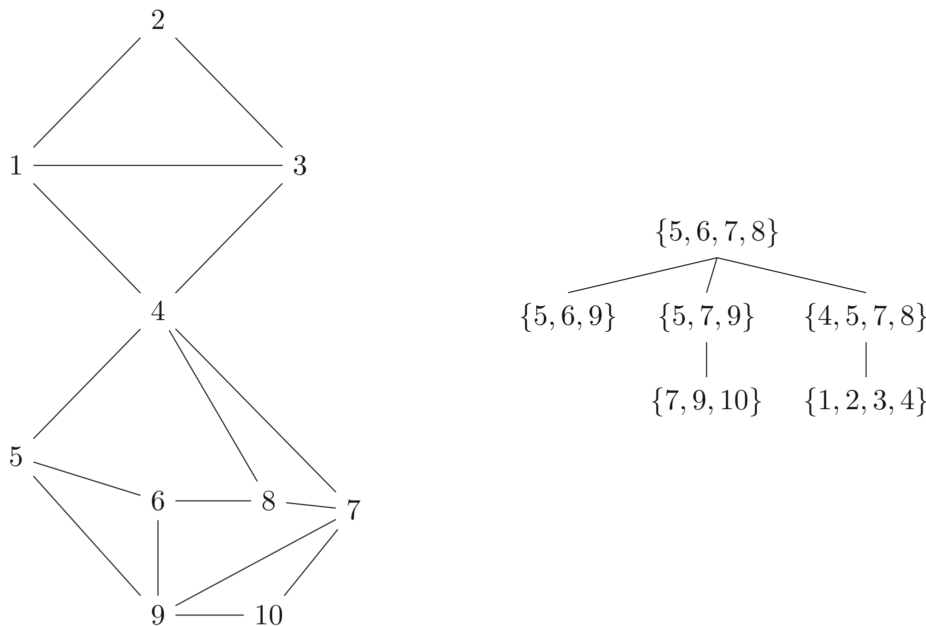


Figure 3.1: A graph and a tree decomposition constructed from the elimination ordering $(1, 2, 3, 4, 8, 6, 5, 9, 7, 10)$

3.7 Further Approaches

An interesting technique for computing FHDs was developed by Fichte et al. (2018), where the problem of finding an FHD is encoded as an SMT problem and solved using the Z3 solver. Schidler and Szeider (2020) extended the encoding to add the descendant condition and compute HDs. The resulting approach proved to be quite effective, performing similar to a version of new-det-k-decomp in the PACE 2019 competition [Dzulfikar et al., 2019].

3.8 Hypergraph Benchmarks

After several rounds of improvement in decomposition algorithms and many different approaches to the problem over the past 20 years, it became apparent that a benchmark suite would be of high utility. Different systems for computing decomposition often focused either on database query (CQ) instances or on CSP instances; consequently, it was hard to judge how they performed on the other types of queries. Comparing different systems in a comprehensive manner, covering all types of hypergraphs having various properties and widths is challenging without standard benchmarks [Gottlob et al., 2020a].

To tackle this issue, Fischl et al. (2019) have developed a set of benchmarks named *HyperBench*. It consists of 3070 hypergraph instances from CQ and CSP problems as well as randomly generated instances. All HyperBench instances, together with their properties, and as well for many instances, HD/GHD/FHD solutions, are available

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

online[3]. The hypergraphs were aggregated from several sources, some from pre-existing benchmarks, some extracted from other sources, and some randomly generated, leading to a set of problems with a broad range of properties.

Approximately 500 instances were generated with a random query generator, which has the advantage of being able to produce instances of high width, as opposed to most real-world instances. Out of a large dataset of over 26 million SPARQL queries, almost all turned out to be acyclic. 70 of $hw \geq 2$ were extracted and included in the benchmark. Only 8 of them turned out to have a hypertree-width of 3. Some queries based on the frequently-used TPC-H benchmark [Transaction Processing Performance Council, 2014] were also included. SQLShare [Jain et al., 2016] is an online software as a service for managing and sharing research data, based relational databases and SQL. Out of the queries made by users of SQLShare, trivial cases and redundancies were eliminated and 270 queries taken into HyperBench. Out of the HyperBench CSP instances, the vast majority (1953) were taken from the XCSP[4] instances. XCSP is an XML-based format for expressing CSP problems with a large number of associated instances, of various algorithmic problems, such as Sudoku, scheduling problems or coloring problems.

Fischl et al. (2019) carried out an empirical analysis of the hypergraph parameters in the HyperBench instances. They considered the *degree*, *intersection width*, *multi-intersection-width*, and *VC-dimension*, comparing them between *application-CQs*, *random-CQs*, *application-CSPs*, and *random-CSPs*, which turned out to have quite different properties.

Application-CQs and application-CSPs had by far the "nicest" properties. For both, over 90% of the instances fulfill the 2-BIP, and for both of them the VC-dimension is bounded by 3, and for the application-CQs, it is even bounded by 2. The degree of all application-instances is also bounded by 6 in the majority of cases. Random-CQs and random-CSPs are quite similar in their properties. The degree of random-instances is, on average, significantly higher than that of non-random instances, with over 95% of the random-CSPs having a degree greater than 5.

These unfavourable properties make random-CSPs and CQs particularly challenging to work with. On the other hand, this can be seen positively, as the vast majority of instances extracted from real-world applications are, in fact, easy to decompose.

An interesting result that could be empirically investigated by finally having an efficient method for computing optimal GHDs was that the ghw corresponds to or almost corresponds to the hw in most cases. On the HyperBench data, in 99% of the fully solved instances, the ghw is identical to the hw, and in all other instances, the ghw is only greater by 1.

[3]http://hyperbench.dbai.tuwien.ac.at
[4]http://xcsp.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

CHAPTER 4

Query Processing

In this chapter, the foundations of queries and query processing are introduced. We start with the foundations behind our optimization approach: conjunctive queries, acyclicity of queries, and Yannakakis' algorithm. Then, we explain how database systems optimize queries and make use of statistics for optimization.

4.1 Conjunctive Queries

Conjunctive Queries (CQs) are a fundamental formalism for expressing queries over relational databases. A *database* is a set of *relations*, each consisting of a name, a *schema*, and a set of *tuples* (i.e. rows). The *schema* defines a list of *attributes* (i.e. columns), which each have an associated *domain*, a set of possible values [Chandra and Merlin, 1977].

We will now consider a simple relational database schema based on the TPC-H benchmark [Transaction Processing Performance Council, 2014]:

```
region(regionkey)
nation(regionkey, nationkey, name)
customer(nationkey, custkey)
supplier(nationkey, suppkey)
lineitem(suppkey, orderkey, discount, extendedprice)
orders(orderkey, custkey)
```

CQs can be seen as functions mapping a database to another database and selecting a set of attributes and their associated values. They consist of a head and a conjunction of atoms over a subset of the relations. For example, the following CQ over the schema above joins the `nation`, `customer`, and `orders` relations, outputting the `nation.name` (N) and `orders.orderkey` (O) attributes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$$Q_1 = \{(N, O) \mid \exists R, A, C : \texttt{nation}(R, A, N) \land \texttt{customer}(A, C) \land \texttt{orders}(O, C)\} \tag{4.1}$$

Figure 4.1: A conjunctive query over 3 relations with 2 joins

A common alternative syntax for CQs is the datalog notation:

$$Q_1(N, O) \leftarrow \texttt{nation}(R, A, N), \texttt{customer}(A, C), \texttt{orders}(O, C). \tag{4.2}$$

If the head of a CQ is empty (i.e. there are no free variables), it is referred to as a *Boolean Conjunctive Query (BCQ)*. Consider for example the boolean version of our query above:

$$\{() \mid \exists R, A, C, N, O : \texttt{nation}(R, A, N) \land \texttt{customer}(A, C) \land \texttt{orders}(O, C)\} \tag{4.3}$$

A BCQ does not answer the question *which sets of tuples fulfill the conditions?* as CQs do but *is there a set of tuples fulfilling the conditions?* Hence, answering a BCQ only requires finding a set of tuples in the tables that match instead of enumerating all answers. BCQs are a well-studied subset of CQs because of their simplicity and results with respect BCQs often allowing generalization to CQs [Gottlob et al., 2002].

Two CQs Q_1 and Q_2 are said to be *equivalent* if, for any database instance D, the results $Q_1(D)$ and $Q_2(D)$ ($Q_1 \equiv Q_2$) are equivalent. Likewise, Q_1 is *contained in* Q_2 ($Q_1 \subseteq Q_2$) if, for any database D, $Q_1(D) \subseteq Q_2(D)$ [Chandra and Merlin, 1977]. These definitions give rise to the associated decision problems *query-equivalence* and *query-containment*. From query-containment, query-equivalence can trivially be derived; hence the focus of complexity results is more frequently on query-containment. There are immediate consequences to the problem of *query-optimization*, which essentially deals with finding an equivalent query of better or optimal performance and thus relies on checking query-equivalence. The decision problems of *boolean conjunctive query evaluation (BQE)*, *tuple-of-query (QOT)* (checking whether a tuple is contained in the output of a CQ), and *query-containment* are all NP-complete [Chandra and Merlin, 1977; Gottlob et al., 2002].

Because the problems associated with CQs, in general, take as input both a query and a database, a frequent distinction is made between the *query complexity* and the *data complexity* [Vardi, 1982]. The *combined complexity* refers to the complexity of a problem (e.g., BQE) where both a database D as well as a query Q are considered as input. In the case of *query complexity*, only the query Q is part of the input, with the database fixed. *Data complexity* considers the query fixed and only the database D as a variable input. In real-world situations, the data complexity is usually more relevant, since query size tends to be constant while it is of interest how the system scales with increasing data.

The extraction of the hypergraph structure of a CQ can easily be done by considering each atom of the query's body as a hyperedge and the variables of the atoms as vertices.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We denote the hypergraph of a query as $\mathcal{H}(Q)$. Our running example query Q_1 gives rise to the following hypergraph:

$$\mathcal{H}(Q_1) = (V, E) \text{ with } V = \{A, C, N, O, R\} \text{ and } E = \{\{R, A, N\}, \{A, C\}, \{O, C\}\} \quad (4.4)$$

Much of the interest in CQs comes from the fact that SQL *select-from-where* statements with equality conditions are directly equivalent. A join query equivalent to our CQ is given in Figure 4.2. Since they do not increase the expressive power of an SQL expression, we will ignore aliases, explicit `JOIN` clauses and other syntactic variants for simplicity. In order to convert an SQL statement to a CQ, we take the projection after the `SELECT` clause as the head of the CQ and the relations from the `FROM` clause as the body, with the variable structure decided by the join statements in the `WHERE` clause. The variables of the CQ are then determined by forming equivalence classes from the equality conditions of the SQL query. In the query of Figure 4.2, two equivalence classes are formed: 1) line 3 gives rise to the CQ variable A, corresponding to the columns `nation.nationkey` and `customer.nationkey` 2) line 4 gives rise to the variable C, corresponding to the columns `customer.custkey` and `orders.custkey`

```sql
SELECT name, orderkey
FROM nation, customer, orders
WHERE nation.nationkey = customer.nationkey
AND customer.custkey = orders.custkey
```

Figure 4.2: An SQL query corresponding to the CQ Q_1

4.2 Acyclic Conjunctive Queries

Acyclic conjunctive queries (ACQs) are a simple, easy to solve class of queries because they coincide with the class of queries for which a *join tree* exists, making them efficient to evaluate [Beeri et al., 1981; Yannakakis, 1981; Gottlob et al., 2001a]. A query Q is *acyclic* if its associated hypergraph $\mathcal{H}(Q)$ is acyclic. There are multiple notions of acyclicity in hypergraphs, and we here refer by acyclicity to *α-acyclicity*, the most general notion of acyclicity [Fagin, 1983]. The most intuitive characterization of *alpha*-acyclicity is the property of having a join tree.

Definition 4.2.1 (Join Tree)**.** A *join tree* of a hypergraph $\mathcal{H}$ is a tree $T = (N, E)$ with $N = E(\mathcal{H})$ where for each hypergraph vertex $v \in V(\mathcal{H})$, $\{e \in N \mid v \in e\}$ induces a connected subtree.

The join tree of a query Q corresponds to the join tree of its hypergraph $\mathcal{H}(Q)$.

A query Q is α-acyclic if a join tree for its hypergraph $\mathcal{H}(Q)$ exists.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Example 4.2.1. A possible join tree of Q_1 (Figure 4.1) is:

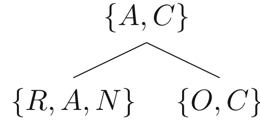


Figure 4.3

Therefore, Q_1 is α-acyclic.

Example 4.2.2. Consider the query $Q_2(D, C) \leftarrow \texttt{supplier}(N, S) \wedge \texttt{customer}(N, C) \wedge \texttt{orders}(O, C) \wedge \texttt{lineitem}(S, O, D, E)$.

$\mathcal{H}(Q_2) = (\{C, D, E, N, O, S\}, \{supplier = \{N, S\}, customer = \{N, C\}, orders = \{O, C\}, lineitem = \{S, O, D, E\}\})$.

The hyperedges $\{supplier, customer, orders, lineitem\}$ are joined in a cycle. Therefore, it would not be possible to find a join tree of Q_2 due to the connectedness condition and acyclicity of trees.

We will now present an algorithm for deciding if a query is acyclic and finding a join tree at the same time: the *GYO reduction* was introduced by Graham (1980), Yu and Özsoyoglu (1979).

Definition 4.2.2. The *Graham-Yu-Öszoyoğlu Reduction (GYO-reduction)* is a nondeterministic algorithm which applies the following two operations exhaustively (as a don't-care nondeterminism) on a hypergraph:

1) Remove hyperedges that are empty or contained in another hyperedge

2) Remove vertices that are contained in at most one hyperedge

$GYO(\mathcal{H})$ refers to the hypergraph obtained after performing the GYO-reduction on $\mathcal{H}$.

A hypergraph $\mathcal{H}$ is acyclic and has a join tree if and only if $E(GYO(\mathcal{H})) = \emptyset$. From the GYO edge elimination ordering we can afterwards construct a join tree.

Example 4.2.3. A possible sequence of GYO reduction operations on $\mathcal{H}(Q_1) = (V, E)$ with $V = \{A, C, N, O, R\}$ and $E = \{\{R, A, N\}, \{A, C\}, \{O, C\}\}$:

1. operation 2: remove $\{R, N\}$ from hyperedge $\{R, A, N\}$ and O from hyperedge $\{O, C\}$: $\mathcal{H}(Q_1) = (\{A, C, N, O, R\}, \{\{A\}, \{A, C\}, \{C\}\}$
2. operation 1: remove hyperedges $\{A\}$ and $\{C\}$: $\mathcal{H}(Q_1) = (\{A, C, N, O, R\}, \{\{A, C\}\}$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. operation 2: remove $\{A, C\}$ from hyperedge $\{A, C\}$: $\mathcal{H}(Q_1) = (\{A, C, N, O, R\}, \{\{\}\}$
4. operation 1: remove hyperedge $\{\}$: $\mathcal{H}(Q_1) = (\{A, C, N, O, R\}, \{\}$

From the parent-child relationships induced by the application of operation 1, we can construct the join tree seen in Figure 4.3. Since the hyperedges $\{R, A, N\}$ and $\{O, C\}$ are eliminated due to containment in $\{A, C\}$, they are added as children of $\{A, C\}$ in the join tree.

Example 4.2.4. An attempt to perform the GYO reduction on the cyclic query Q_2 i.e., its hypergraph $\mathcal{H}(Q_2)$ will, after one application of rule 2, terminate with $GYO(\mathcal{H}(Q_2)) = (\{C, D, E, N, O, S\}, \{\{N, S\}, \{N, C\}, \{O, C\}, \{S, O\}\})$.

Due to the fact that not all edges of $GYO(\mathcal{H}(Q_2))$ were eliminated, Q_2 is a cyclic query.

The GYO reduction runs in time quadratic in the size of the hypergraph. However, it can be checked efficiently, in time $O(|V| + |E|)$, whether a hypergraph is acyclic, by using the algorithm presented by Tarjan and Yannakakis (1984).

Acyclic queries are widespread in real-world databases and applications, and the polynomial complexity of evaluating them using their join trees makes them unproblematic. Nevertheless, many queries are not strictly acyclic and, going by our approach of constructing a join tree, would seemingly leave no other choice than to perform a full join between all tables at once, leading to a significant blowup of intermediate results. The consequence of this would be a major computational effort in the worst case.

This issue motivated the generalization of α-acyclicity to larger classes of queries in order to also bound their complexity. Hypertree-width is such a generalization of α-acyclicity. The class of queries having hypertree-width 1 corresponds to the class of ACQs, and, with increasing width, the degree of cyclicity increases. HDs also correspond to join trees, where an HD of width 1 directly corresponds to a join tree as defined above and an HD of $hw > 1$ can be converted to a join tree by joining the λ bags.

Example 4.2.5. A width 1 HD of query Q_1 and a width 2 HD of query Q_2. Both are minimal and the first is also a join tree.

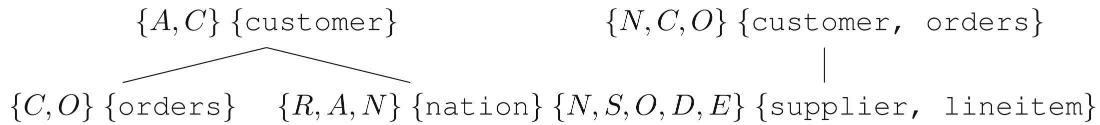
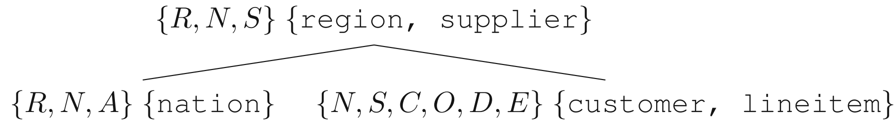


Figure 4.4

Example 4.2.6. Consider the query

$$Q_3 \leftarrow \texttt{region}(R) \wedge \texttt{nation}(R, N, A) \wedge \texttt{supplier}(N, S) \wedge \texttt{customer}(N, C) \wedge \texttt{orders}(O, C) \wedge \texttt{lineitem}(S, O, D, E) \tag{4.5}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



A minimal HD of Q_3 is:

4.3 Yannakakis' Algorithm

The algorithm introduced by Yannakakis (1981) showed that acyclic BCQs can be solved in polynomial time, and the enumeration of answers to ACQs is possible in output polynomial time. In Algorithm 10, the standard version of Yannakakis' algorithm for BCQs evaluation is given. Given a query Q, its join tree $T = (V, E)$ and a database D, it returns *true* if the BCQ is satisfied. A bottom-up traversal is done along the join tree. Each join tree node with children is semi-joined with its children, sorting out all tuples which do not have a matching join-partner in at least one of the children. In the end, due to the bottom-up semi-joins, every join relation was effectively semi-joined with all relations below it. If the root relation has a non-empty result, we can conclude that the result set of the full join cannot be empty and, consequently, the BCQ is satisfied.

Algorithm 10 Yannakakis' algorithm for BCQs

```
C(n) denotes the child nodes of a tree node n ∈ V
procedure YANNAKAKIS(Q, T = (V, E), D)
    for all n ∈ V with C(n) = {c_1, ..., c_k}, k ≥ 1 and |C(c_i)| = 0 for 0 ≤ i ≤ k do
        let R_n be the relation in D associated with node n
        for all child nodes c ∈ C(n) do
            let R_c be the relation in D associated with node c
            R_n := R_n ⋉ S_c
            remove c from V
        end for
    end for
end procedure
procedure EVALUATEBCQ(Q, T = (V, E), D)
    YANNAKAKIS(Q, T, D)
    return true if R_r ≠ ∅, else return false, where r is the root node of T
end procedure
```

In order to retrieve the full result set of the acyclic query, two more passes over the relations are required (Algorithm 11): after the first bottom-up round of semi-joins, the same process is applied top-down. Finally, the remaining data in all relations is joined together in a full join.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

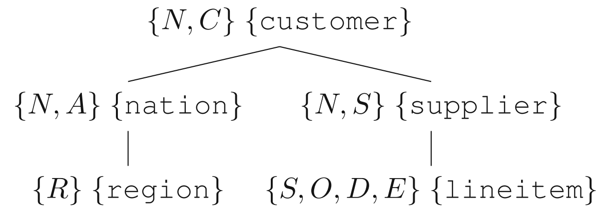
Algorithm 11 Yannakakis' algorithm for ACQs

procedure EVALUATEACQ($Q, T = (V, E), D$)
  $T' :=$ invert T
  YANNAKAKIS(Q, T, D)
  YANNAKAKIS(Q, T', D)
  return $\bowtie_{R_i \in D} R_i$
end procedure

Example 4.3.1. Consider the query Q_4 (obtained from Q_3 by breaking the cycle and removing `orders`):

$$Q_4 \leftarrow \texttt{region}(R) \wedge \texttt{nation}(R, N, A) \wedge \texttt{supplier}(N, S) \wedge \texttt{customer}(N, C) \wedge \texttt{lineitem}(S, O, D, E) \quad (4.6)$$

Q_4 is acyclic and the following join tree (T_4) exists:



Let D be a database with the following relations and rows[1]:

region
regionkey
r_1
r_2
r_3

nation		
regionkey	nationkey	name
r_1	n_1	argentina
r_3	n_6	germany
r_3	n_7	france
r_3	n_{19}	romania

customer	
nationkey	custkey
n_1	c_1
n_1	c_2
n_6	c_3
n_7	c_4
n_8	c_5

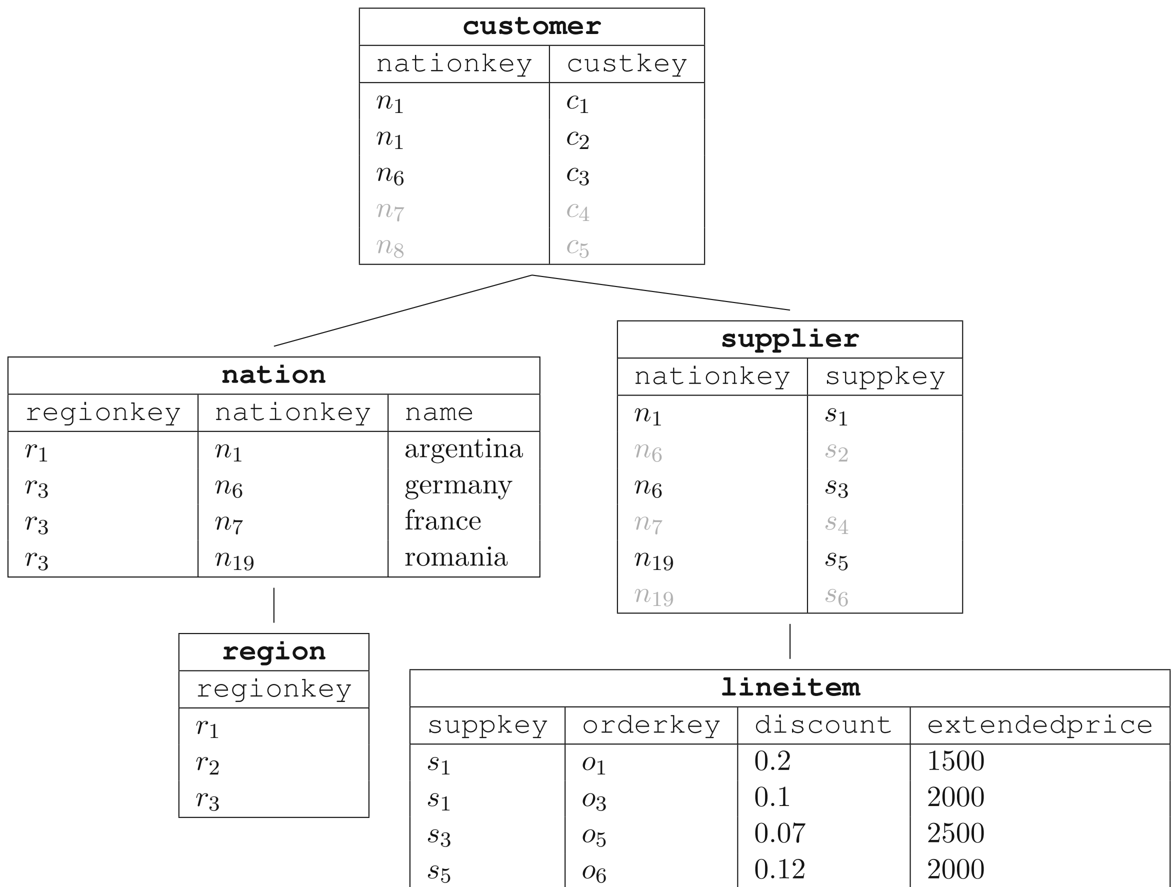
supplier	
nationkey	suppkey
n_1	s_1
n_6	s_2
n_6	s_3
n_7	s_4
n_{19}	s_5
n_{19}	s_6

[1] The data was partially taken from the autogenerated TPC-H [Transaction Processing Performance Council, 2014] tables and the rest constructed for this example. Note that referential integrity was violated for providing a better example of the algorithm.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

lineitem			
suppkey	orderkey	discount	extendedprice
s_1	o_1	0.2	1500
s_1	o_3	0.1	2000
s_3	o_5	0.07	2500
s_5	o_6	0.12	2000

Now, we can evaluate the boolean query Q_4 along the join tree T_4 by applying Yannakakis' algorithm:



customer	
nationkey	custkey
n_1	c_1
n_1	c_2
n_6	c_3
n_7	c_4
n_8	c_5

nation		
regionkey	nationkey	name
r_1	n_1	argentina
r_3	n_6	germany
r_3	n_7	france
r_3	n_{19}	romania

supplier	
nationkey	suppkey
n_1	s_1
n_6	s_2
n_6	s_3
n_7	s_4
n_{19}	s_5
n_{19}	s_6

region
regionkey
r_1
r_2
r_3

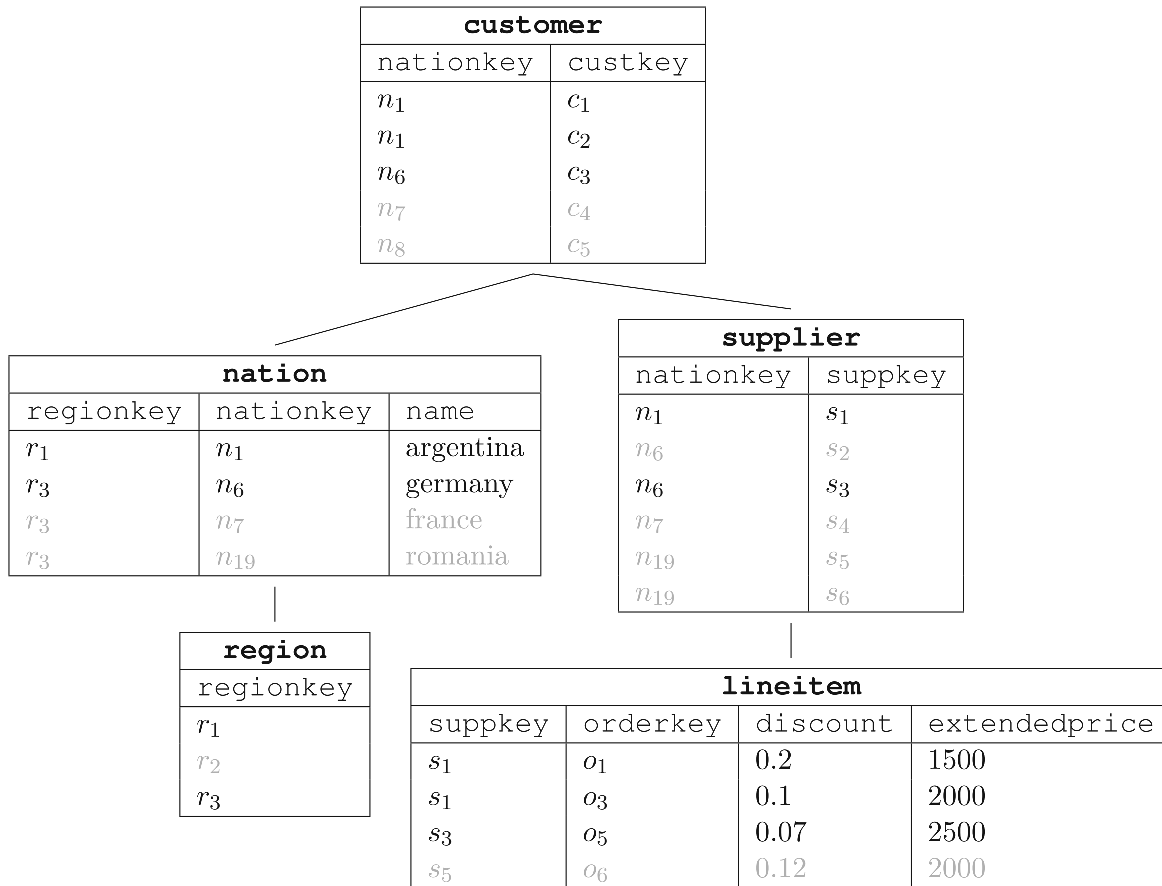
lineitem			
suppkey	orderkey	discount	extendedprice
s_1	o_1	0.2	1500
s_1	o_3	0.1	2000
s_3	o_5	0.07	2500
s_5	o_6	0.12	2000

At the root node, the three rows $\{(n_1, c_1), (n_1, c_2), (n_6, c_3)\}$ remain after semi-joining. Therefore, the boolean query Q_4 is satisfied. Considering a non-boolean variant of Q_4

$$Q_4'(regionkey, name, discount) \leftarrow \texttt{region}(R) \wedge \texttt{nation}(R, N, A) \wedge \texttt{supplier}(N, S) \wedge \texttt{customer}(N, C) \wedge \texttt{lineitem}(S, O, D, E) \tag{4.7}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

we can now enumerate all answers to this query. A second pass over the join tree is done, this time from the top down:



customer	
nationkey	custkey
n_1	c_1
n_1	c_2
n_6	c_3
n_7	c_4
n_8	c_5

nation		
regionkey	nationkey	name
r_1	n_1	argentina
r_3	n_6	germany
r_3	n_7	france
r_3	n_{19}	romania

supplier	
nationkey	suppkey
n_1	s_1
n_6	s_2
n_6	s_3
n_7	s_4
n_{19}	s_5
n_{19}	s_6

region
regionkey
r_1
r_2
r_3

lineitem			
suppkey	orderkey	discount	extendedprice
s_1	o_1	0.2	1500
s_1	o_3	0.1	2000
s_3	o_5	0.07	2500
s_5	o_6	0.12	2000

Finally, all relations are joined together and projected:

$$\pi_{\texttt{regionkey,name,discount}}(\texttt{region} \bowtie \texttt{nation} \bowtie \texttt{customer} \bowtie \texttt{supplier} \bowtie \texttt{lineitem}) \tag{4.8}$$

We obtain the result set:

regionkey	name	discount
r_1	argentina	0.2
r_1	argentina	0.1
r_3	germany	0.07

Yannakakis' algorithm is sequential. Still, computing a join tree of an acyclic BCQ is in the class LogCFL, as is the problem of evaluating an acyclic BCQ with respect to a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

join tree, therefore making the combined problem of evaluating an arbitrary acyclic BCQ fall into LogCFL [Gottlob et al., 2001a]. Thus, in theory, acyclic BCQ evaluation is also a highly parallelizable problem. Yannakakis' algorithm can as well be parallelized, however only as far as the structure of the tree permits, where the number of tree nodes per layer influences the maximum extent of parallelization.

Hypertree decompositions are not the only decomposition method that can be used to generalize acyclicity. Alternatively, the tree-width of the hypergraph, query-width or hinge-width are also valid generalizations. One question that immediately arises is: *what makes hypertree-width a better generalization than other methods?*. Gottlob et al. (2016) identified 3 conditions that a generalization of acyclicity has to fulfill in order to be successfully applied to query-answering:

1. **Generalization of Acyclicity:** Queries of width ≥ 1 should include all acyclic queries
2. **Tractable Recognizability:** Queries of width k can be recognized efficiently
3. **Tractable Query-Answering:** Bounded-width queries can be answered efficiently

Evaluating other decomposition techniques by these criteria, tree-width and hinge-width fail by violating condition 1. Query-width, while being a true generalization of acyclicity, fails at condition 2 since queries of bounded query-width are not tractably recognizable.

4.4 Query Optimization

Relational database systems are expected to provide fast answers to as many queries as possible. Finding the fastest way to execute a query submitted in a query language such as SQL is a challenging task of database systems. The cause of the complexity of this task are the various factors affecting a query execution's runtime in practice, the large search space of possible query executions and the problem of estimating the performance of an execution plan.

Queries are specified by the user in a relational query language. In most widely-used systems this language is SQL. The first step is to parse the query language statement into a tree of *relational algebra operations* [Codd, 1970]. A *query plan* describes how the DBMS will execute the query. It is usually structured in the form of a tree of data-manipulating operators based on relational algebra expressions mixed with lower-level details. For example, a query plan might be made up of (index) scans, various kinds of joins, sorting, projections, filters, set operations or even decisions such as storing data to disk (*materialization*) and network transfer. Hence, query optimization in practice is a more complex process than just finding an equivalent query, with respect to relational algebra. The *query optimizer* component of the DBMS has the task of coming up with an effective query plan, which the *query executor* interprets or executes [Chaudhuri, 1998].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

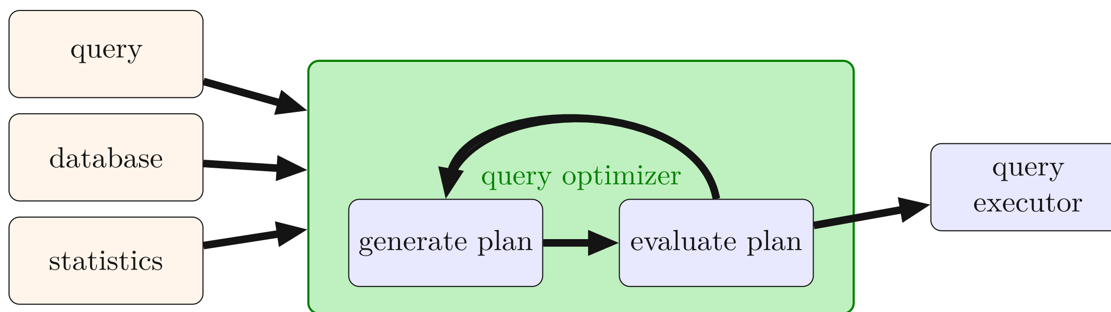


Figure 4.5: Query optimization and execution in DBMS

Various properties of the data and of the system will have an influence on the effectiveness of an individual query plan. Physical data storage, and especially the choice of index (e.g., a B-Tree, or a hash table) versus unindexed data can make a vast difference, reducing polynomial runtime query plans to linear or linearithmic time. While the structure of the query itself restricts the space of valid join trees, the data contained in the database has a significant influence on the runtime of a query plan, due to the differences in intermediate results produced in the joins.

The strategy of *cost-based optimization* was originally introduced by IBM [Astrahan et al., 1975; Selinger et al., 1979] as part of the System R database system prototype, which originated in a research project and included the first implementation of SQL. Its architecture and design choices influenced most later relational database systems and are still found in state-of-the-art systems.

Two major tasks are solved by the query optimizer: the *enumeration of plans* and the *evaluation of plans*. Due to the exponential number of query plans with a rising number of relations, an exhaustive enumeration tends to become infeasible at a point. Therefore, on complex queries, query optimizers employ heuristics to restrict the search space and make the search for good query plans feasible. As valid by the relational algebra equivalences, optimizers always apply the simple but highly effective heuristics of pushing down projections and selections in the tree. These heuristics have the effect of reducing the size of intermediate results and improving many query plans significantly. The problems of finding an effective join order and join algorithm selection thus remain the greater challenges [Deshpande et al., 2007].

The evaluation of query plans is perhaps the most important procedure of the optimizer, since an overestimation of the performance of query plans would lead to the acceptance of sub-optimal plans. In this procedure, first, *selectivities* of the join plan are calculated, which are estimations of the number of rows produced by each operator. Next, the operator costs can be estimated by calculating the number of expected rows per operator.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.5 Join Algorithms

There are three basic join algorithms with different performance and use-cases [Astrahan et al., 1975]. We restrict ourselves to the standard case of non-distributed database systems and consider relations with bag semantics (i.e., duplicates may occur). All join algorithms for evaluating $R \bowtie S$ have a worst-case output-size and runtime of $O(|R| \cdot |S|)$, which occurs in the case that the join degenerates to a cross-product due to the input data. By extension, through repeated application of binary joins, evaluating a join-tree of relations $R_1, ..., R_n$ produces in the worst-case an output of size $O(|R_1| \cdots |R_n|)$.

Note that in practice, especially in the case of HDs, disk block access plays a major role in the performance of join algorithms. For simplicity, block handling is ignored and costs are considered only in terms of tuple access.

4.5.1 Nested-Loop Join

The *nested-loop join* is the naive approach towards join processing. It involves enumerating all pairs of tuples and checking for matching join attributes.

A join $R \bowtie_C S$ with a boolean condition $C : (r \in R, s \in S) \rightarrow \{0, 1\}$ is evaluated the following way using the nested-loop algorithm:

Algorithm 12 Nested-Loop Join

```
procedure NestedLoopJoin(R, S, C)
    for all r in R do
        for all s in S do
            if C(r, s) is fulfilled then
                emit (r, s)
            end if
        end for
    end for
end procedure
```

The cost of evaluating a binary nested-loop join is $\Theta(|R| \cdot |S|)$. Due to the quadratic runtime over all databases, plain nested-loop joins are rarely chosen by database systems for evaluating joins. Nested-loop joins tend to be the fastest option in the case explicit cross-products are wanted, due to no sorting or hash-table buildup overhead. Nevertheless, a variant known as *index nested-loop join* is commonly used if indexes are available. By replacing the inner loop with an index lookup, the complexity is reduced to $O(|R| \cdot \log(|S|))$ on relations without duplicates if a B-Tree index is available on one relation (S).

In the case of semi-joins ($R \ltimes_C S$), the asymptotic cost of evaluation stays the same but the average runtime will be lower in practice, since not all of the tuples contained in the second relation need to be checked.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 13 Nested-Loop Semi-Join

```
procedure NestedLoopSemiJoin(R, S, C)
    for all r in R do
        for all s in S do
            if C(r, s) is fulfilled then
                emit r
                break
            end if
        end for
    end for
end procedure
```

4.5.2 Merge Join

The *merge join* makes use of the sortedness of two relations to join their tuples efficiently. If the tables need to be sorted beforehand, the algorithm is referred to as a *sort-merge join*. The sorting step is the most expensive and its performance depends on the sorting algorithm used. In the case a worst-case $O(n \log n)$ sorting algorithm (e.g. mergesort) is used to sort the relations, the total sort-merge runtime is at worst $O(|R| \cdot \log |R| + |S| \cdot \log |S|)$. Comparatively, the cost of merging is low at $O(|R| + |S|)$, hence the sorting step is significantly more expensive. In practice, (sort-)merge-joins tend to be more rarely used by PostgreSQL than the other variants.

4.5.3 Hash Join

The *hash join* is an approach to replace the inner loop of the join with a constant-effort lookup procedure. It can only be applied to equi-joins of the form $R \bowtie_C S$. We can consider C as a set of equality conditions between attributes $\{(x, y) | x \in \mathit{schema}(R), y \in \mathit{schema}(S)\}$. First, a hash table of the (typically) smaller relation (let us assume $|R| > |S|$) is built up. The join attributes of the tuple serve as the hash key and are mapped to a list of the corresponding complete tuples.

Algorithm 14 Hash-Join

```
procedure HashJoin(R, S, C = {(x_1, y_1), ..., (x_n, y_n)})
    build up a hash table H : (a_1, ..., a_n) → S
    for all r in R do
        for all s in H(r[x_1], ..., r[x_n]) do
            emit (r, s)
        end for
    end for
end procedure
```

Evaluating a natural hash join without duplicates can be done in time $O(|R| + |S|)$. In

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the worst case of a cross product, the runtime is again $\Theta(|R| \cdot |S|)$, due to the result set size. However, in the case of a semi-join with duplicates, and hence also in the general case, the worst-case complexity stays at $O(|R| + |S|)$. Hash joins tend to be the most effective join strategy if indexes are not available, however they are restricted to equi-joins. Otherwise, the DBMS needs to make use of nested-loop joins or merge joins [Khayyat et al., 2015]. Furthermore, if the hash table (i.e. the smaller relation) is too large to fit in memory, due to the tendency of the hash join to become slower than the merge join, the latter will be chosen by the DBMS.

4.6 Query Optimization in PostgreSQL

PostgreSQL was one of the earliest open-source database systems [Stonebraker and Rowe, 1986; Stonebraker et al., 1990] and is still one of the major two open-source systems in use at the time of writing (together with MySQL). Its optimizer was originally based on the System R optimizer and has not changed significantly. The standard configuration of PostgreSQL is a single-server setup without load-balancing or partitioning. A *master-process* spawns a new process for each established *connection*. Data integrity between connections and transactions is guaranteed via inter-process communication through semaphores and shared memory. In general, query execution inside a connection is single-threaded. Parallel query plans are possible but rarely chosen by the optimizer in PostgreSQL 13. At least partially, this is due to the multi-process architecture, which causes the parallelization and communication overhead to be relatively large. Both the constant cost of setting up parallelization (`parallel_setup_cost`) and the cost of transferring a tuple (`parallel_tuple_cost`) are set to a constant default value but are configurable, influencing the choice of parallel plans.

After establishing a connection, a client can send queries to the server. Queries are transmitted as plain-text SQL strings and parsed into a syntax tree, then passed to the query optimizer (referred to as *planner/optimizer* in PostgreSQL). Since it can be done mostly independently from join order optimization, the optimizer starts by optimizing the table scan types. The choice of table scans depends on the available indexes and the attributes and operators occurring in the query. Afterwards, the search for a good join tree is started.

4.6.1 Exhaustive Search Optimization

PostgreSQL implements two different approaches towards join optimization depending on the query's complexity, as measured by the number of relations in the query. If a query has fewer tables than the value of the parameter `geqo_threshold` (default: 12), the *(near-)exhaustive search* algorithm is applied; otherwise the *genetic query optimizer (GEQO)* is used.

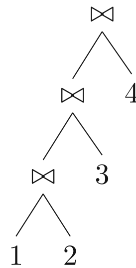
The *(near-)exhaustive search* algorithm enumerates almost all possible (binary) join trees and chooses the one estimated to be the least costly. It is not fully exhaustive since trees

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

with cross-products are excluded from the search space, significantly reducing the search space at a small cost of potentially missing better query plans in rare cases [Leis et al., 2015]. Join tree nodes may use the join strategies *(index-)nested-loop join*, *(sort-)merge join*, or *hash join*.

4.6.2 Genetic Query Optimization

Due to the size of the query execution search space growing exponentially with the number of tables joined, exhaustively searching all join trees becomes more expensive than executing the query. The Institute of Automatic Control at the University of Mining and Technology (Freiberg, Germany) encountered issues using PostgreSQL for decision-support queries in a large knowledge-based system Utesch (1999). To answer these queries efficiently, a heuristic search procedure in the form of a genetic algorithm was developed. It is designed to take over query optimization at the point where the exhaustive search optimizer becomes too expensive. Left-deep join trees are encoded as strings representing join paths (*chromosomes/genes* in the context of GAs), encoding the problem like a TSP (Travelling Salesperson Problem). For example, the string `1-2-3-4` represents the join tree where first relations 1 and 2 are joined, then the result is joined with relation 3 and finally with relation 4:



The initial population of join orderings is randomly generated. Join orderings are rated by their estimated costs using the same function as in the exhaustive search procedure. A steady-state genetic algorithm is employed, where in each iteration the worst join orderings are replaced by orderings created through recombination of the previous generation. The *edge-recombination operator* is employed, a mutation operator designed for recombining paths and commonly used for solving TSP problems, which also proved to be effective for join order optimization, despite it being a different problem (Postgres, 2020, geqo.html).

4.6.3 Query Plans and Cost Estimation

A *query plan* in PostgreSQL is a tree of *plan nodes*. *Scan nodes* are located at the leaves of the tree, reading data from disk and returning tuples. The other nodes consist of join, aggregation, sorting, and further data-manipulating operations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For an example of a query plan, consider the query over the TPC-H database (generated with a size of 0.6GB) joining 6 tables (see Figure 4.6), and its query plan (see Figure 4.7) generated by PostgreSQL 13.

```sql
SELECT nation.n_name, lineitem.l_extendedprice, lineitem.l_discount
FROM customer, orders, lineitem, supplier, nation, region
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND l_suppkey = s_suppkey
AND c_nationkey = s_nationkey
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey;
```

Figure 4.6: An SQL query over the TPC-H database

```
Hash Join  (cost=56088.85..273487.66 rows=1153173 width=38)
  Hash Cond: (customer.c_nationkey = nation.n_nationkey)
  ->  Gather  (cost=56083.02..259547.66 rows=384391 width=20)
        Workers Planned: 2
        ->  Parallel Hash Join  (cost=55083.02..220108.56 rows=160163 width=20)
              Hash Cond: ((orders.o_custkey = customer.c_custkey) AND (supplier.s_nationkey = customer.c_nationkey))
              ->  Hash Join  (cost=37830.82..172570.10 rows=1500388 width=20)
                    Hash Cond: (lineitem.l_suppkey = supplier.s_suppkey)
                    ->  Parallel Hash Join  (cost=37294.83..151403.77 rows=1500388 width=20)
                          Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
                          ->  Parallel Seq Scan on lineitem  (cost=0.00..82508.88 rows=1500388 width=20)
                          ->  Parallel Hash  (cost=28317.48..28317.48 rows=547148 width=8)
                                ->  Parallel Seq Scan on orders  (cost=0.00..28317.48 rows=547148 width=8)
                    ->  Hash  (cost=386.55..386.55 rows=11955 width=8)
                          ->  Seq Scan on supplier  (cost=0.00..386.55 rows=11955 width=8)
              ->  Parallel Hash  (cost=13470.08..13470.08 rows=200008 width=8)
                    ->  Parallel Seq Scan on customer  (cost=0.00..13470.08 rows=200008 width=8)
  ->  Hash  (cost=4.89..4.89 rows=75 width=30)
        ->  Hash Join  (cost=1.11..4.89 rows=75 width=30)
              Hash Cond: (nation.n_regionkey = region.r_regionkey)
              ->  Seq Scan on nation  (cost=0.00..2.75 rows=75 width=34)
              ->  Hash  (cost=1.05..1.05 rows=5 width=4)
                    ->  Seq Scan on region  (cost=0.00..1.05 rows=5 width=4)
```

Figure 4.7: The query plan of the query in Figure 4.6

As shown in Figure 4.7, each plan node is labeled with its estimated costs in arbitrary units and output sizes depending on the cost parameters. The first cost value represents the estimated cost until the output of rows can start (*startup-cost*), and the second represents the estimated total cost. In the top-level node, the startup-cost is at 56088 only about 20% of the total cost of 273487, as expected due to the join operation, where most of the effort is in the join loop. The number of rows is estimated to be 1,153,173 and the average size in bytes per row is estimated to be 38; hence, PostgreSQL estimates the total output size to be approximately 44MB (Postgres, 2020, using-explain.html).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

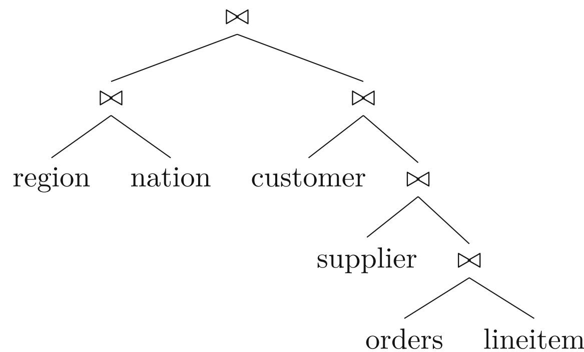


Figure 4.8: The join tree of the query plan in Figure 4.7

The query plan cost estimation in PostgreSQL is relatively complex, taking into account several statistics. Per default, PostgreSQL 13 only makes use of *single-column statistics*, PostgreSQL has to make independence assumptions about the columns. However, in cases where column values are correlated, these assumptions can lead to incorrect estimations and inefficient query plans. Therefore, PostgreSQL also makes it possible to compute *extended statistics* (multivariate statistics) on demand.

Basic metadata about relations as well as the estimated number of rows and pages of a relation is stored in the `pg_class` system catalog, and managed by PostgreSQL. The approximate values tend to become out-of-date as data is manipulated and can be explicitly updated using the VACUUM and ANALYZE commands (Postgres, 2020, catalog-pg-class.html).

Single-column statistics of all relations are stored in the `pg_statistic` catalog. For security reasons, such that it is not possible to extract information from the tables of other users, `pg_statistic` is access-restricted to the superuser. `pg_stats` is a more readable view of the low-level data in `pg_statistic`, and is readable by all users, while only allowing the user to view their own tables (Postgres, 2020, planner-stats.html). Similarly, if available, the extended statistics are stored in the system catalogs `pg_statistic_ext` and `pg_statistic_ext_data`, and can be read by all users through the view `pg_stats_ext`.

`pg_stats` contains a row for each attribute, in each relation of each database (Postgres, 2020, view-pg-stats.html). The maximum number of entries of the most common values (MCVs) and histogram can be configured, and is as of PostgreSQL 13 set to 100. This way, the tradeoff between optimization overhead and performance can be controlled.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

name	**type**	**description**
null_frac	float $[0, 1]$	fraction of null values
avg_width	int > 0	size in bytes per row
n_distinct	float	number of distinct values or distinct values per row if < 0
most_common_vals	array	list of the most common values
most_common_freqs	float array	the frequencies of the most common values
histogram_bounds	array	list dividing the column's values into an equi-depth histogram
correlation	float $[-1, 1]$	correlation between physical and logical ordering
most_common_elems	array	as most_common_vals for non-scalar types (e.g. arrays)
most_common_elem_freqs	float array	as most_common_freqs for non-scalar types
elem_count_histogram	float array	as histogram_bounds for non-scalar types

Figure 4.9: Single-column statistics estimated by PostgreSQL

PostgreSQL gathers all statistics via sampling. Hence they are only estimates, and will vary with repeated sampling. Selectivity estimation is done, ignoring the case of non-scalar types, with the MCVs (`most_common_vals`), distinct counts (`n_distinct`), and the histogram (`histogram_bounds`). The *null fraction* (`null_frac`) is used for adjusting selectivity estimations in columns with null values and the `correlation` value is used to estimate the cost of scan operations.

We will not go into selectivity estimation in full detail, since the process involves many technicalities and case distinctions. For a more comprehensive overview, the reader is referred to the PostgreSQL documentation and the source code[2].

Histograms are important for estimating selectivities involving numeric values and inequality operations. In fact, histograms are only computed by PostgreSQL if the column type supports inequality comparisons.

Example 4.6.1 (Histograms)**.** Consider a relation `rel` with two numeric floating point attributes `a` and `b`:

a	**b**
1	2
2.5	3
0.5	0.8
4	5
8	9
3	3
1	6
1.5	7
0.7	5
4	8.5

[2]The selectivity estimations are implemented in `src/backend/utils/adt/selfuncs.c`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The following two equi-depth (each bin contains approximately the same number of entries) histograms with 5 bins might be constructed by PostgreSQL:

a	b
0.5 0.7	0.8 2
1 1	3 3
1.5 2.5	5 5
3 4	6 7
4 8	8.5 9

Assume we want to evaluate the query **SELECT** a,b **FROM** rel **WHERE** a $<$ 3 **AND** b $<$ 3. For relation a, 3/5 bins contain values less than 3 and, for relation b, one bin does. Thus, assuming independence between the columns, PostgreSQL would estimate the selectivity to be $\frac{3}{5} \cdot \frac{1}{5}$ by multiplying the selectivities of the conditions. Assuming an estimated count of 10,000 rows, the row count would be estimated to be 1200.

For equality conditions and equi-joins, histograms are not very useful. In these cases, the optimizer relies on the MCV list instead. Since this list can by default grow to up to 100 entries, PostgreSQL tends to be able to relatively accurately estimate the selectivities of joins producing large result sets.

We will now consider the selectivity estimation of equi-joins. If MCV lists are available for both joined attributes, the lists are matched up. For two relations r_1 and r_2, let M_1 and M_2 be their MCV sets, let N_1 and N_2 be their null fractions and let D_1 and D_2 be the number of distinct values. $F_1 : M_1 \to [0,1]$ and $F_2 : M_2 \to [0,1]$ map the MCVs to their frequencies. The frequencies of MCVs occurring in both relations are multiplied and summed up:

$$C_{match} = \sum_{v \in M_1 \cap M_2} F_1(v) \cdot F_2(v) \tag{4.9}$$

Next, we can calculate the frequency of non-null values that are not in the MCV lists:

$$\begin{aligned} U_1 &= 1 - N_1 - |M_1| \\ U_2 &= 1 - N_2 - |M_2| \end{aligned} \tag{4.10}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Now, we can estimate the selectivity of the unmatched MCVs of relation i with non-MCV values of relation j:

$$C^{i,j}_{unmatch,uncommon} = \frac{|M_i \setminus M_j| \cdot U_j}{D_j - |M_j|} \tag{4.11}$$

We estimate the selectivity of non-MCV values in relation i with non-MCV values in relation j and unmatched MCV values in relation j.

$$C^{i,j}_{uncommon} = \frac{U_i \cdot (U_j + |M_j \setminus M_i|)}{|D_j| - |M_i \cap M_j|} \tag{4.12}$$

Finally, we can determine the total selectivity:

$$C_{total} = C_{match} + min(C^{1,2}_{unmatch,uncommon} + C^{1,2}_{uncommon}, C^{2,1}_{unmatch,uncommon} + C^{2,1}_{uncommon}) \tag{4.13}$$

If at least one of the MCV lists is missing, another calculation based on the distinct values is used for estimating the join selectivity:

$$C_{total} = min(\frac{1}{D_1}, \frac{1}{D_2}) \cdot (1 - N_1) \cdot (1 - N_2) \tag{4.14}$$

4.6.4 Optimizer Configuration

While PostgreSQL does not support optimizer hints as other database systems do, it provides various parameters that can be configured to change its behaviour. By setting the boolean parameters beginning with `enable_` to false, the space of possible plans considered by the optimizer can be restricted, for example, disabling merge joins (`enable_mergejoin`) or sorting (`enable_sort`).

The query optimizer makes use of several cost constants when rating query plans, which represent characteristics of the system, such as the CPU speed (`cpu_tuple_cost`), random disk access (`random_page_cost`) and sequential disk access (`seq_page_cost`). It is also possible to configure parallelism by setting the number of parallel workers and the assumed costs of parallel processing.

Nested join queries are often written for readability, and one might expect the subqueries to be optimized independently from the rest of the query. This would, however, lead to a restriction of the search space, therefore, PostgreSQL flattens subqueries (as well as explicit joins). Flattening of subqueries is controlled by the configurable parameter `from_collapse_limit`: if the total number of FROM conditions in all subqueries is not more than this number, the query is flattened. By default, this value is set to 8 FROM relations (Postgres, 2020, explicit-joins.html).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Integrating Structural Decompositions into Database Systems

After briefly going over the state of the art of structural-decomposition-based optimization, we give an overview of our new system and explain the details of the query execution pipeline and weighted decompositions. Furthermore, we explain how hypergraphs can be stored and which approach we used for visualizing them.

5.1 State of the Art

Optimization techniques based on structural decomposition have not yet found their way into mainstream open-source and commercial database systems. Several research prototypes have, however, been developed which make use of structural decompositions for query answering, with promising results.

5.1.1 H-DB

The H-DB system developed by Ghionna et al. (2007, 2011) was, to the best of our knowledge, the first and only attempt to integrate structural decompositions into a DBMS. Unfortunately, the source-code of the H-DB system is no longer available, making it impossible to get access to the implementation details and the exact benchmarking data for comparison with the system developed in this thesis.

H-DB is implemented as a graphical application, which allows the user to connect to a database server and enter queries. User-entered queries are first parsed to verify syntactic correctness and to extract the join structure as a hypergraph. Next, the statistics required

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for optimization are extracted from the PostgreSQL system catalogs. The hypergraph and associated table statistics (a weighted hypergraph) are passed to the `cost-k-decomp` decomposition algorithm [Scarcello et al., 2007].

The resulting hypertree decomposition is finally translated into an execution plan and executed in PostgreSQL with the help of custom modifications of the optimizer. We are however left to speculate how exactly the integration was done.

From their experiments with H-DB, the authors come to the conclusion that solely using minimal-width structural decompositions for optimization was not competitive against statistical optimization. However, the structural decomposition approach combined with statistics was able to beat the standard PostgreSQL optimizer in their experiments, surprisingly even on the TPC-H benchmark.

5.1.2 EmptyHeaded

EmptyHeaded is a *high-level graph engine* developed by Aberger et al. (2017) and based on the ideas of Tu and Ré (2015). It follows a very different approach to standard database systems and implements *worst-case optimal joins* [Ngo et al., 2013].

As shown by Atserias et al. (2008) (AGM bound), the number of output tuples of a conjunctive query can be tightly bounded, depending on the fractional hypertree width. Additionally, by only considering two-way joins, any possible query execution is asymptotically suboptimal. The typical example of this is the *triangle query* $R \bowtie S \bowtie T$ over three relations $R(A, B)$, $S(B, C)$, and $T(A, C)$ with $|R| = |S| = |T| = N$. Intuitively, since any two of the relations' attributes/values fully cover those of the third relation, the output size and thus the worst-case complexity can be bounded to N^2, however by applying the AGM bound, it can be reduced to $O(N^{\frac{3}{2}})$. For two-way joins, $\Omega(N^2)$ is indeed the limit. Nonetheless, the worst-case join algorithm by Ngo et al. (2013) can evaluate the triangle query in $O(N^{\frac{3}{2}})$.

The query language is based on datalog and supports aggregation operations as well as a restricted form of recursion. Data is stored in tries. GHDs are used as query plans and Yannakakis' algorithm is utilized for query answering at the level of the decomposition. To find minimal-width GHDs, an exhaustive search is done. Inside decomposition bags, the worst-case optimal algorithm is applied, which is implemented using multi-way join operators. Due to the heavy use of set intersections in the worst-case optimal joins, these were highly optimized using SIMD (simultaneous computation of multiple values in a single CPU instruction) operations.

In their experiments, the authors showed that EmptyHeaded outperformed other graph analytics by 4x-60x and LogicBlox by over three orders of magnitude. EmptyHeaded differs significantly from conventional database systems due to its use of multi-way join. Integrating worst-case optimal joins similar to EmptyHeaded into PostgreSQL would require completely restructuring its query optimization and execution engines.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

LogicBlox is a commercial system for data analytics developed by Aref et al. (2015)[1], making use of decomposition-based query optimization. It comes with the query language *LogiQL*, based on datalog and featuring various aggregation operations. Its major intended use case is the analysis and prediction of sales data. The source code of LogicBlox is not open and there are few implementation details of how structural decomposition techniques are used.

5.2 Overview of our System

We have developed a lightweight system to automatically optimize SQL queries transparently to the user without modifying the underlying database system. Our main goal was to determinine whether such a lightweight integration is feasible and can lead to better results than the standard PostgreSQL optimizer. Furthermore, we investigated the parallelization of query plans based on hypertree decompositions.

Most applications utilizing relational databases for storing and querying persistent data directly connect to a DBMS server using a standard protocol. Furthermore, Java applications make use of the JDBC standard interface, thus adding a layer of abstraction and allowing the application developers to change the database without changing the code in theory. In practice, JDBC is still a leaky abstraction, since databases do not all support the same features and behave differently.

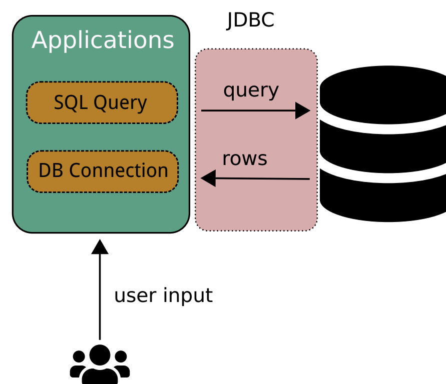


Figure 5.1: JDBC applications[2]

Our query optimization and execution system is a proxy layer between the application and the database, which implements a JDBC-like interface as a Java library. Although not implemented as a fully transparent proxy, the integration of the system can be done with few modifications to the existing code.

[1] https://developer.logicblox.com/
[2] Icons licensed under CC Attribution 4.0 (https://fontawesome.com/license)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

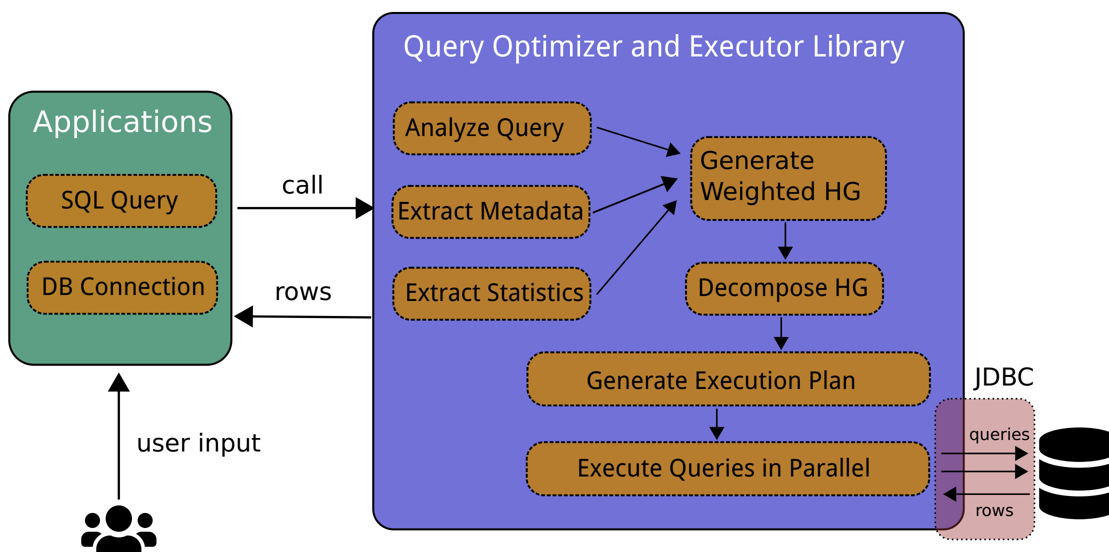


Figure 5.2: The system architecture

Client applications need to pass a JDBC connection (i.e., a pool of connections for parallelization) and a query to the optimizer/executor library, which will do the rest of the work transparently to the client and return the result set after completion.

The library extracts the necessary metadata from the database to become aware of the existing relations, and parses the given SQL query to convert it to a hypergraph. Additionally, it extracts statistics and adds weights to the hypergraph. The weighted hypergraph is then decomposed, optimizing for hypertree-width first and weight second. This decomposition tree is used for executing Yannakakis' algorithm. Several parallel steps of multiple queries along the tree are generated, where intermediate results are stored in temporary tables, and then executed in parallel.

5.3 Weighted Hypertree Decompositions

Although the application of (G)HDs and Yannakakis' algorithm can theoretically lead to significant speedups, in practice, join queries between multiple tables tend to be significantly skewed, making it necessary to integrate statistics for reliable performance.

Example 5.3.1. Consider a cyclic query (a triangle query joined with an additional relation) with 4 relations $R(a, b)$, $S(b, c)$, $T(a, c)$, $U(a, x)$:

```
SELECT *
FROM R
NATURAL JOIN S
NATURAL JOIN T
NATURAL JOIN U;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

join	selectivity
$R \bowtie S$	0.01
$R \bowtie T$	0.01
$S \bowtie T$	0.01
$U \bowtie R$	0.01
$U \bowtie S$	1
$U \bowtie T$	0.01

Additionally, assume $|R| = |S| = |T| = |U| = 1000$ and the following join selectivities:

All join selectivities are relatively low, except that of the join $U \bowtie S$, due to it being a cross-product. If we want to evaluate this join by applying Yannakakis' algorithm, we first have to decompose the hypergraph. Two of the possible decompositions, which might be the output of BalancedGo, are:

Figure 5.3

Solely from the point of view of hypertree-width, these decompositions are equally good. By applying the heuristic that fewer expensive natural joins are better, decomposition T_1 appears to be the better choice. However, considering the selectivities and join costs, T_1 is a worse choice than T_2.

As can be seen in the following join tree, annotated with the sizes of the (intermediate) tables in the bags, the join between S and U produces an output 100 times larger than the joins $R \bowtie S$ and $T \bowtie U$. Assuming a join strategy of hash joins in the bags and hash semi-joins between the bags, the costs of evaluating T_1 are significantly higher than the costs of evaluating T_2.

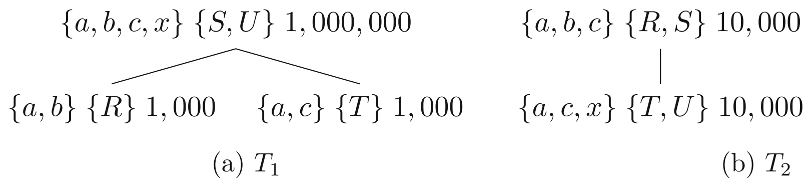


Figure 5.4

Similar to the concepts presented by Scarcello et al. (2007) and applied by Ghionna et al. (2007), we decided to make use of weighted hypertrees and decompositions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We considered a restricted case of weighted hypergraphs, which we will refer to *bag-weighted hypergraphs*. While the concept could be extended to a more general notion of weighted hypergraphs, taking into consideration semi-joins between bags, we decided to only optimize for the sum of bag-joining costs, since it was efficiently implementable in BalancedGo. Taking into consideration semi-joins as well would have caused the search problem to become much harder, since a change of the weight of one bag would require evaluating the whole tree to determine the change in the total cost. This is still an interesting open problem for future research, and would, if solved, improve the effectiveness of the query optimization approach even further.

Definition 5.3.1. A *bag-weighted hypergraph (bwHG)* $(\mathcal{H}, W)$ is a hypergraph $\mathcal{H} = (V, E)$ together with an associated weight function $W : 2^E \rightarrow \mathbb{R}$.

The weight function W is defined for all possible hypertree bags $b \subseteq E$, assigning to them the (estimated) size of the bag after joining or in the case of $|b| = 1$, the size of the relation.

(Generalized) hypertree-width is still used as the primary optimization objective, with the join cost of the decomposition coming second.

Definition 5.3.2. A *bag-weighted generalized hypertree decomposition (bwGHD)* is a GHD (T, χ, λ, W) with $T = (N, E)$ over a bwHG $(\mathcal{H}, W)$.

The weight of a bwGHD is defined as the sum of the bag-weights:

$$weight(G) = \sum_{b \in N} W(\lambda(b)) \quad (5.1)$$

A *width-k minimal bag-weighted generalized hypertree decomposition (k-minimal bwGHD)* is a bwGHD $D = (T, \chi, \lambda, W)$ of a hypergraph $\mathcal{H}$ where (T, χ, λ) is a width-k GHD and $weight(D)$ is minimal over all GHDs of width k of $\mathcal{H}$.

A *minimal bag-weighted generalized hypertree decomposition (minimal bwGHD)* is a bwGHD $D = (T, \chi, \lambda, W)$ is a k-minimal bwGHD where (T, χ, λ) is a minimal-width GHD.

In the optimization system developed in this thesis, we use a heuristic search procedure to look for (close to) minimal bwGHDs. It was implemented as an extension to BalancedGo [Gottlob et al., 2020c][3] The weighted decomposition procedure is based on the `LocalBIP` (see Section 3.4) search procedure. Bags (i.e. separators) are sorted by their join costs and iterated in this order. Consequently, bags of lower join cost are prioritized in the search. As later seen in the experimental part, this hybrid approach of searching for minimal-width GHDs with a minimal total weight, proved to be significantly more effective in practice than considering only width (see chapter 6.2.

[3]The extension is at the time of writing unpublished but available as a branch in the repository: `https://github.com/cem-okulmus/BalancedGo`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

5.4 The Query Optimization/Execution Pipeline

We will now follow the path of a query from the client code up to execution, starting from the call to the library. Consider the standard piece of Java code for establishing a connection and executing a query, found in any JDBC application:

```
Connection conn = DriverManager.getConnection(dbURL, properties);

PreparedStatement ps = conn.prepareStatement(query);
ResultSet rs = ps.executeQuery();
```

We only need to make two minor changes in order to integrate the decomposition-based optimizer. For parallelization, a connection pool is created instead of a single connection (alternatively, a single connection can also be passed to the non-parallel executor). A `QueryExecutor` object is instantiated, wrapping the connection, and can then be used in the code like a JDBC `Connection`:

```
ConnectionPool connPool = new ConnectionPool(dbURL, properties);

PreparedStatement ps = conn.prepareStatement(query);
QueryExecutor optimizedQueryExecutor =
        new ParallelViewQueryExecutor(connPool);

ResultSet rs = optimizedQueryExecutor.execute(query);
```

5.4.1 Class Structure

All code developed for this thesis is open-source and available on GitHub: `https://github.com/arselzer/HTQueryOptimizer`. Note that the implementation will likely change over time. The version of the system described in this thesis is found at the following commit: `https://github.com/arselzer/HTQueryOptimizer/tree/04060b28a8805fa2e6eaecd5f73a225aa1643a43`.

Figure 5.5 shows the `QueryExecutor` interface and its subclasses. The class `UnoptimizedQueryExecutor` executes the queries directly without hypertree-based optimizations, and is used to provide a more generic interface, where implementations can be swapped easily. `TempTableQueryExecutor`, is a concrete class which uses temporary tables to store intermediate data and execute the optimized query. Its subclass, `ParallelTempTableQueryExecutor`, extends the query execution part with a parallel execution implementation.

Figure 5.6 shows in more detail the classes used for representing the schema, hypergraphs, weighted hypergraphs, and join trees. `TempTableQueryExecutor` extracts the database schema from the JDBC connection metadata into a `DBSchema` instance and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

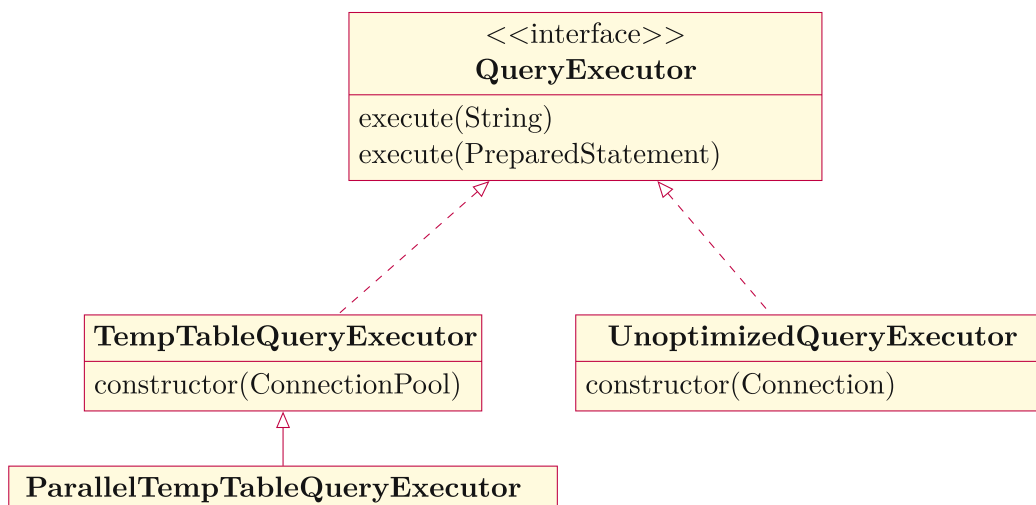


Figure 5.5: `QueryExecutor` class hierarchy

the statistics into a `TableStatistics` instance. The schema and query string are then passed to create an `SQLQuery` instance, representing a query over a database and performing the transformation into a `WeightedHypergraph`. The `WeightedHypergraph` class performs the decomposition procedure up to the transformation to a join tree in the form of a hierarchy of `JoinTreeNode` instances. Finally, the `SQLQuery` instance uses the join tree to construct a `ParallelQueryExecution` instance.

5.4.2 Query to Hypergraph Conversion

The first step of the pipeline involves converting the query input over the database to a hypergraph. In order to interpret the query, the parser first needs to be aware of the database schema. While instantiating the `TempTableQueryExecutor`, an instance of the `DatabaseMetaData` is retrieved from the JDBC connection, from where the tables in the database and their attributes are enumerated and extracted into a `DBSchema` object. This schema is not re-extracted during the lifetime of the `TempTableQueryExecutor` instance for better performance.

Table statistics are extracted on-demand and only for the required tables at the time of executing an SQL query (calling the `execute(String query)` method). For each table joined in the query, the MCVs (see Section 4.6.3) and frequencies are stored.

Before extracting the hypergraph, using the *JSQLParser* library, the projection columns and table aliases are determined. The hypergraph of the query is then determined by a class of the *hgtools* library, which was described by Gottlob et al. (2020a)[4]. From the hypergraph and the statistics, a weighted hypergraph is constructed by iterating all combinations of hyperedges and calculating their costs.

[4]the code of *hgtools* is available at `https://github.com/dmlongo/hgtools`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

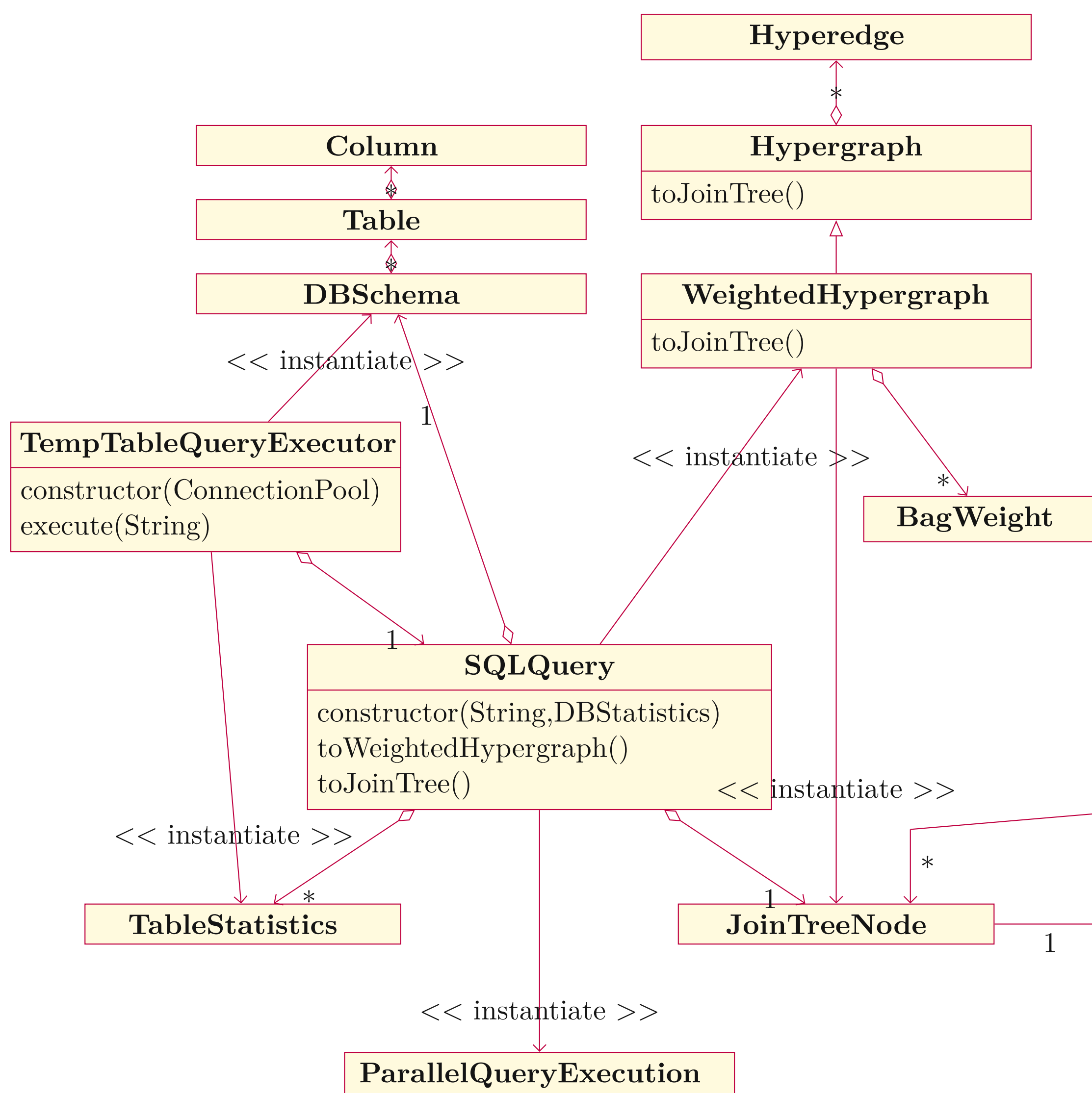


Figure 5.6: `TempTableQueryExecutor` and related classes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
lineitem(X50,X60,X52,X51,X54,X53,X45,X56,X55,X36,X47,X58,X57,X49,X48,X59),
region(X16,X27,X28),
customer(X19,X21,X20,X23,X22,X25,X24,X18),
partsupp(X9,X10,X12,X11,X13),
orders(X8,X45,X2,X3,X4,X5,X18,X6,X7),
part(X41,X40,X43,X42,X44,X36,X38,X37,X39),
supplier(X30,X21,X31,X34,X33,X47,X35),
nation(X21,X16,X15,X17).
```

5.4.3 Hypertree Decomposition

In order to find a close to minimal bwGHD, the hypergraph is written into a file in the HyperBench format (see Section 5.6) and BalancedGo is called with k increasing from 1 until a GHD is returned. The hypertree output of BalancedGo in the GML format is parsed into an in-memory graph structure using the *JGraphT*[5] library. It is then converted into what we will refer to as the *execution tree* (note that the implementing class is named `JoinTreeNode`, although the data structure is not strictly a join tree as defined in Section 4.2). The execution tree differs from the hypertree decomposition as it explicitly contains all hyperedges joined in the query, even in the case one of the hyperedges is not explicitly given in the lambda labels but covered by the other edges.

In order to decrease the size of intermediate results, the execution tree is post-processed by projecting out attributes as far as possible. All attributes not occurring in the parent or child bags, or required for inner-bag joins, are removed, for each bag.

Example 5.4.1. Consider the following query over the TPC-H database:

```sql
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount
from
        customer, orders, lineitem, supplier,
        nation, region, part, partsupp
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and p_partkey = l_partkey;
```

After parsing, it is converted to the following hypergraph (in DTL format):

[5] `https://jgrapht.org/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

l_linenumber	
value	**frequency**
1	0.251966655254364
2	0.21490000188350677
3	0.1783333271741867
4	0.1426333338022232
5	0.10406666994094849
6	0.07029999792575836
7	0.03779999911785126

o_orderstatus	
value	**frequency**
P	0.026100000366568565
F	0.4897666573524475
O	0.4841333329677582

Figure 5.7

MCV lists and frequencies are then extracted from the DB for the relevant attributes. For example, Figure 5.7 shows the MCV lists extracted for two attributes from the TPC-H tables.

relation 1	relation 2	weight
customer	orders	140625000.000
lineitem	region	751950.000
lineitem	orders	5639625000.000
region	nation	125.000
region	partsupp	100000.000
customer	partsupp	75000000.000
orders	nation	937500.000
lineitem	nation	3759750.000
customer	nation	93750.000
partsupp	orders	750000000.000
customer	part	18750000.000
part	supplier	1250000.000
orders	supplier	9375000.000
lineitem	part	751950000.000
region	part	25000.000
supplier	nation	6250.000
region	supplier	1250.000
partsupp	part	100000000.000
lineitem	partsupp	3007800000.000
region	customer	18750.000
lineitem	customer	563962500.000
orders	part	187500000.000
region	orders	187500.000
partsupp	supplier	5000000.000
partsupp	nation	500000.000
part	nation	125000.000
customer	supplier	37386.000
lineitem	supplier	37597500.000

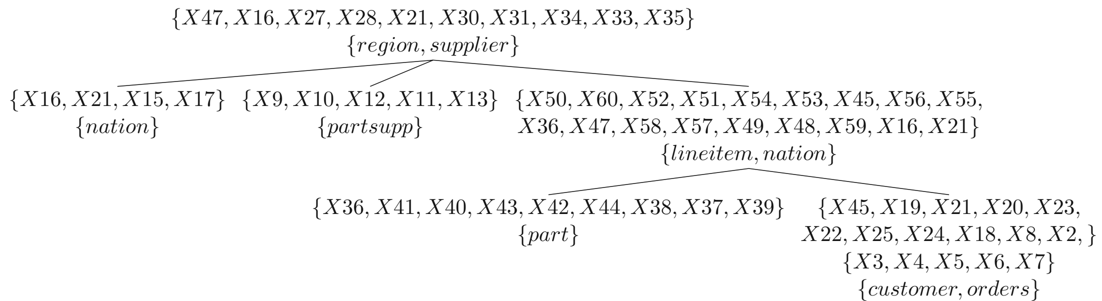
Figure 5.8

Note that in the specific case of the TPC-H DB, the MCV frequencies of the join keys are all equal, therefore we chose a more interesting example. The computed weights based

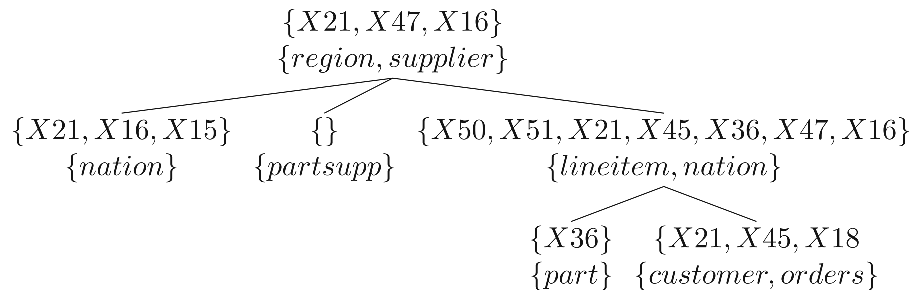
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

on the statistics for all pairs of relations are seen in Figure 5.8.

This weighted hypergraph is decomposed by BalancedGo into the following decomposition:



We can see that numerous attributes are not directly part of the query result or needed for the joins. After projecting out these redundant attributes, the following tree results:



The final result attributes `nation.n_name`, `lineitem.l_extendedprice`, and `lineitem.l_discount` are mapped to the hypergraph vertices $X15$, $X50$, and $X51$, respectively. The tree node containing the relation `partsupp` does not have any attributes, since none are part of the result set and the join with it is a cross-product.

5.5 Optimized Query Execution

Two strategies were implemented for query execution, both based on temporary tables and supporting weighted and unweighted hypergraphs. The first approach, which in general is the better-performing one, is based on the repeated execution of queries creating tables, and parallelization of parallelizable queries. The second approach, execution as a function, creates a PL/pgSQL function, which implements the entire query execution, and then calls the function. It was implemented in such a way in order to decrease the communication overhead of sending multiple queries, however, it has the disadvantage that parallelization is not possible.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.5.1 Parallelized Execution

This execution approach of Yannakakis' algorithm makes use of temporary tables and also runs in parallel for the parts of the algorithm that can be parallelized.

Before execution, the execution tree is converted into an *execution ordering*, a sequence of parallel SQL statements (*layers*), as well as information about the result and the removal of the tables after execution. The execution ordering consists of a sequence of *layers*. These layers are generated in 4 steps (see Section 4.3 on Yannakakis' algorithm):

1. For each execution tree node, if there is more than one table, a natural join is done and the result is stored in a table. Otherwise, a view is created as an alias of the single table. Since this step is fully parallelizable, all queries are part of one layer.
2. Semi-joins are done from the bottom up, and results are stored in tables. The tree nodes are split into sets of the same depth, since all nodes at a certain depth depend on those below them. Queries generating nodes of the same depth are then added to the same layer.
3. Semi-joins are done from the top down, and results are stored in tables. The queries are assigned to layers as in step 2.
4. All tables resulting from step 3 are joined together

A pool of connections is used for query execution parallelism. This is required due to PostgreSQL connections being executed on a single core only. By default, its size is equal to the number of virtual cores of the system. Using multiple connections, however, leads to another problem: temporary tables are no longer shared between queries executed in different connections. Ordinary tables are inefficient for temporary storage due to their use of a WAL (write-ahead log). Therefore, the implementation makes use of *unlogged tables*, which behave like ordinary tables without a WAL. The only significant difference to temporary tables is that unlogged tables are not removed automatically after closing a connection.

Example 5.5.1. Stage 1 of the execution ordering generated from the execution generated from the execution tree of Example 5.4.1 starts with the following four queries:

```
CREATE UNLOGGED TABLE htqo_lineitem_nation_stage_1_93008e
AS SELECT X50, X51, X21, X45, X47, X36, X16 FROM (
  SELECT l_extendedprice AS X50, l_discount AS X51,
  l_orderkey AS X45, l_suppkey AS X47, l_partkey AS X36
  FROM lineitem
) lineitem
NATURAL INNER JOIN
(
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
  SELECT n_nationkey AS X21, n_regionkey AS X16 FROM nation
) nation;

CREATE UNLOGGED TABLE htqo_region_supplier_stage_1_93008e
AS SELECT X21, X47, X16 FROM (
  SELECT r_regionkey AS X16 FROM region
) region
NATURAL INNER JOIN (
  SELECT s_nationkey AS X21, s_suppkey AS X47 FROM supplier
) supplier;

CREATE UNLOGGED TABLE htqo_customer_orders_stage_1_93008e
AS SELECT X21, X45, X18 FROM (
  SELECT c_nationkey AS X21, c_custkey AS X18 FROM customer
) customer
NATURAL INNER JOIN (
  SELECT o_orderkey AS X45, o_custkey AS X18 FROM orders
) orders;

CREATE VIEW htqo_part_stage_1_93008e
AS SELECT X36 FROM (
  SELECT p_partkey AS X36 FROM part
) part;
```

The first three perform the joins of the relations in the lambda labels lineitem $\bowtie$ nation, region $\bowtie$ supplier, and customer $\bowtie$ orders while the last one is simply an alias for part.

In the bottom-up stage, semi-joins such as the following, where the {customer, orders} bag is joined with the {part} bag, are applied:

```
CREATE UNLOGGED TABLE htqo_lineitem_nation_stage_2_93008e
AS SELECT *
FROM htqo_lineitem_nation_stage_1_93008e
WHERE (
  EXISTS (
    SELECT 1 FROM htqo_customer_orders_stage_1_93008e
    WHERE (htqo_lineitem_nation_stage_1_93008e.X21 =
     htqo_customer_orders_stage_1_93008e.X21)
    AND (htqo_lineitem_nation_stage_1_93008e.X45 =
     htqo_customer_orders_stage_1_93008e.X45)))
  AND (EXISTS (
    SELECT 1 FROM htqo_part_stage_1_93008e
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
    WHERE (htqo_lineitem_nation_stage_1_93008e.X36 =
     htqo_part_stage_1_93008e.X36)
  )
);
```

After the bottom-up and top-down stages the final join is done. Due to the explicit renaming of columns to the hypergraph variable names, a natural join over all tables suffices.

```
CREATE VIEW htqo_3675b5703f6941618f8341f1ee5b04ee AS
  SELECT X15, X50, X51
  FROM htqo_region_supplier_stage_3_93008e
  NATURAL INNER JOIN htqo_nation_stage_3_93008e
  NATURAL INNER JOIN htqo_partsupp_stage_3_93008e
  NATURAL INNER JOIN htqo_lineitem_nation_stage_3_93008e
  NATURAL INNER JOIN htqo_customer_orders_stage_3_93008e
  NATURAL INNER JOIN htqo_part_stage_3_93008e;
```

5.5.2 Execution as a PL/pgSQL Function

PL/pgSQL is a procedural language used for implementing functions in PostgreSQL databases. Functions are able to query and manipulate data. In our case, we make use of a PL/pgSQL function for querying data only, since we need a way to execute a series of queries inside PostgreSQL and store intermediate data - a problem that could not be expressed as a single query.

This implementation makes use of the same execution ordering from the parallel implementation. All queries are packed into a PL/pgSQL function, which is then generated by the query executor and called.

5.5.3 Boolean Query Execution

Both query execution methods can also execute the queries as boolean queries, i.e. queries where the answer is the presence or absence of at least one row. The query optimization and execution procedure is reduced to the first step of bottom-up semi-joins (see Section 5.5.1) and limited to one row at the end by appending a **LIMIT** 1.

5.6 Hypergraph and Hypertree Formats

The most common way of representing hypergraphs is what we will refer to as the *HyperBench format* [Fischl, 2018]. It is a simple plain-text format in which a list of hyperedges is specified, and vertices are given implicitly. This allows expressing any hypergraphs except cases where a vertex is isolated, which are usually not of interest

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

since they cannot result from the conversion of CQs. In Figure 5.9, an example of this format can be seen.

```
HE0(0,1,2),
HE1(1,3,4),
HE2(3,5,6),
HE3(7,8,2),
HE4(8,9,4),
HE5(9,10,6).
```

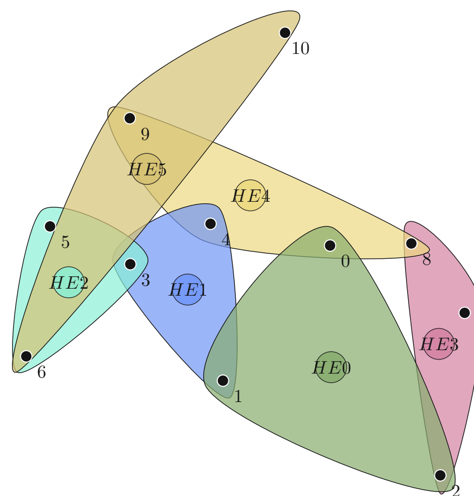


Figure 5.9: A hypergraph (instance 2212 from HyperBench) in *HyperBench format*

An alternative but similar format was used for the PACE 2019 competition.[6] It allows representing isolated vertices, but is less readable than the HyperBench format since the hyperedge names are enumerated. Consequently, it would also be inconvenient for visualization since the hyperedge names, which in our application correspond to table names, are not directly associated with the hyperedges anymore.

```
c Lines beginning with c are comments
c This hypergraph has 11 vertices and 6 edges
p htd 11 6
1 1 2 3
2 2 4 5
3 4 6 7
4 8 9 3
5 9 10 5
6 10 11 6
```

Figure 5.10: The hypergraph from Figure 5.9 in the *PACE format*

Hypertree decompositions are slightly more complex to represent in a file. A common approach is to save them in the *GML (Graph Modelling Language) format*, which is the output of BalancedGo and new-det-k-decomp. GML itself is a standard format for

[6] https://pacechallenge.org/2019/htd/htd_format/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

representing graphs, and in our case, the tree is represented as an undirected graph with the λ and χ sets as part of the vertex labels. An example of the format is seen in Figure 5.11. Parsing the hypertree is more difficult, and best done with an existing library, such as JGraphT for Java (used in our system)[7].

```
graph [
directed 0
node [
id 18
label "{HE0, HE4} {0, 1, 2, 4, 8, 9}"
vgj [
labelPosition "in"
shape "Rectangle"
]]
node [
id 19
label "{HE3} {2, 7, 8}"
vgj [
labelPosition "in"
shape "Rectangle"
]]
node [
id 20
label "{HE1, HE5} {1, 3, 4, 6, 9, 10}"
vgj [
labelPosition "in"
shape "Rectangle"
]]
node [
id 21
label "{HE1, HE2} {1, 3, 4, 5, 6}"
vgj [
labelPosition "in"
shape "Rectangle"
]]
edge [
source 18
target 19
]
edge [
source 18
target 20
]
edge [
source 20
target 21
]]
```

Figure 5.11: A minimal-width GHD in *GML format* of the hypergraph in Figure 5.9, generated by BalancedGo

[7] https://jgrapht.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

An alternative to the GML format is the PACE 2019 hypertree-format (see Figure 5.12), which was specified exclusively for representing hypertrees, and is easier to parse without using a library. However, it is also harder to read and redundant since, for each bag, the hyperedges not part of it also have to be explicitly stated. The lines of the form "`w` a b c" represent whether (as shown by c, which is 0 or 1) vertex b is contained in bag a.

```
c A hypertree with 4 bags, width 2 of a hypergraph
c with 11 vertices and 6 edges
s htd 4 2 11 6
b 1 1 2 3 5 9 10
b 2 3 8 9
b 3 4 5 7 10 11
b 4 4 5 6 7
w 1 1 1
w 1 2 0
w 1 3 0
w 1 4 0
w 1 5 1
w 1 6 0
w 2 1 0
w 2 2 0
w 2 3 0
w 2 4 1
w 2 5 0
w 2 6 0
w 3 1 0
w 3 2 1
w 3 3 0
w 3 4 0
w 3 5 0
w 3 6 1
w 4 1 0
w 4 2 1
w 4 3 1
w 4 4 0
w 4 5 0
w 4 6 0
```

Figure 5.12: The GHD from Figure 5.11 in the *PACE format*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.7 Hypergraph Visualization

Visualizing binary graphs is a very active field of research, with numerous tools for drawing graphs in various ways and having certain properties. However, currently, there are still few tools for visualizing hypergraphs. In the course of working on this thesis, it became apparent that an automatic visualization tool for hypergraphs would be useful.

Several approaches exist for drawing hypergraphs, with some trade-offs [Paquette and Tokuyasu, 2011]:

1. *Primal graph*: In this simple approach, the primal graph (the binary graph constructed by creating cliques from hyperedges) of the hypergraph is generated, and then standard graph-drawing tools are used to draw it. All edges associated with a hyperedge have to be colored in order to avoid losing hyperedge information. However, hyperedges fully covered by other hyperedges cannot be visualized.

2. *Covering vertices by hyperedges*: The most frequent approach for visualizing hypergraphs in the literature. However, hypergraphs presented in this format are often drawn manually, which is a time-consuming process. Drawing such graphs in a visually clear way, which requires them to fulfill various geometric constraints, appears to be a difficult problems. To the author of this thesis, no tool for drawing hypergraphs in this format is known at the time of writing.

3. *Hyperedges as vertices*: Also referred to as the incidence graph. For each hyperedge, a visually distinct vertex is created and connected to all vertices contained in the hyperedge.

4. *PAOH (Parallel Aggregated Ordered Hypergraph)* [Valdivia et al., 2021]: Vertices are arranged as rows and hyperedges as columns. The hyperedge columns "connect" all vertices contained in the hyperedge and hyperedges are sorted in a way that gives a clear overview of the data. Using this representation, large hypergraphs can be visualized effectively. However, it might not give the reader such a good visual idea of the structure of smaller hypergraphs as the other methods.

For this thesis, a randomized algorithm was developed for drawing graphs in the *covering vertices by hypergraphs* representation. While relatively inefficient and its visualizations breaking down with larger hypergraphs, the tool proved to be surprisingly effective for visualizing small to medium-sized hypergraphs.

Drawing candidates are represented as a set of vertex positions (x and y coordinates). On the top level, 1000 drawing candidates are generated. A function rates the candidates. According to the following conditions, points are deducted:

- Closeness of vertices to other hyperedges (increases with distance)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- A vertex is contained graphically in another hyperedge it is not covered by formally (discouraged by a very high constant - this should not happen and leads to inaccurate drawings but might still be better than no drawing)
- The ratio of the longer side of the rectangle containing the hyperedge to its shorter side to avoid "long and thin" edges (increasing with higher ratio)
- The ratio of the longest distance between two neighbour vertices along the convex hull of the hyperedge to the shortest to encourage evenly-spaced outer vertices (increasing with higher ratio)
- The minimum angle between two vertices along the convex hull to encourage "round" polygons (increasing with decreasing angle)

The generation of drawing candidates is described in Algorithm 15. After generating a number of candidates, the best-rated is chosen for drawing. Vertices are drawn on the specified coordinates and a smoothed-curve polygon is drawn along the outside of the convex hulls of the hyperedges, transparently, in order to allow overlapping hyperedges. Binary edges are simply connected by straight line segments, as in a classical binary graph drawing, leading to better scalability as they take up less space than curves. The final drawing is produced by generating LaTeX TikZ commands and then rendering them to a PDF. This way, it would be possible to take an automatically generated hypergraph drawing and manually modify it as needed easily.

If the size of the graph is too large, drawings with significant overlappings can occur, requiring multiple attempts and manual selection. The algorithm could still be enhanced significantly, leading to better runtime and improved visualizations. For example, for drawing binary edges, an existing well-performing graph drawing approach could be used, or a heuristic search technique employed for iteratively improving the search for drawing candidates, instead of generating a fixed number at the start and choosing the best one.

For some examples of visualized hypergraphs, see Appendix B, where the benchmarking queries and their hypergraphs are shown.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

Algorithm 15 generateDrawing

```
Configurable Parameters: NewPointRadius, MaxAttempts, RequiredVertexDistance
procedure GENERATEDRAWING(H = (V, E))
    drawnVertices ← {}
    for all e ∈ E do
        for all v ∈ e do
            newEdgeList ← {}
            pointValid ← false
            attempts ← 0
            vertexAttempt ← null
            while ¬pointValid ∧ attempts < MaxAttempts do
                lastVertex ← the last vertex added to newEdgeList
                vertexAttempt ← (x, y) with random coordinates x and y within NewPointRadius of lastVertex
                minVertexDistance ← the distance from vertexAttempt to the closest vertex in drawnVertices
                if minVertexDistance < RequiredVertexDistance then
                    if e is a binary edge or does not cross any other binary edges then
                        pointValid ← true
                    end if
                end if
                attempts ← attempts + 1
            end while
            add vertexAttempt to drawnVertices and newEdgeList
        end for
    end for
    return drawnVertices
end procedure
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Experimental Evaluation

6.1 Benchmarking

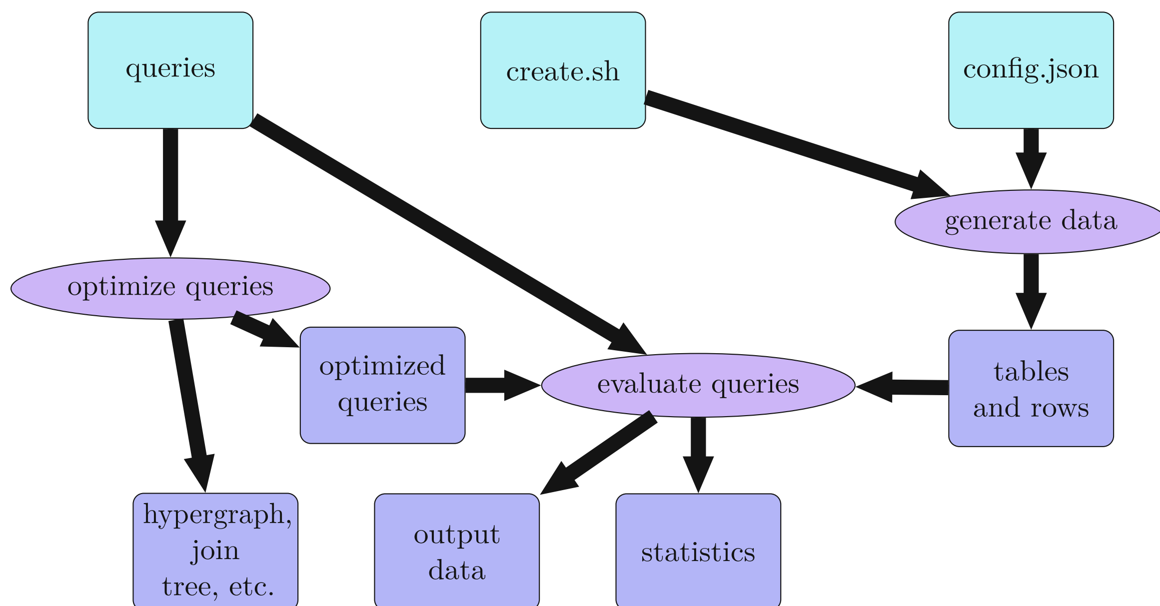


Figure 6.1: The benchmarking pipeline

For the evaluation of the newly developed decomposition-based optimizer against the standard PostgreSQL optimizer, a benchmarking system was developed. It comprises benchmark configuration, data generation, query execution, and correctness checking. The benchmarking instances are is given as a set of *queries* over a set of *databases*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Databases specify the schema and how data is generated. All scripts to generate the data can be found in the git repository[1].

6.1.1 Methodology

An overview of the benchmarking pipeline is given in Figure 6.1. The benchmarking tool is implemented as a Java commandline utility. Its configuration is done through commandline arguments (for details see the README in the repository) and by reading a directory structure with configuration files. The tool enumerates all directories under *data/*, which represent the databases. In each database directory, a setup script named *create.sh* is placed, which creates the database schema and inserts the data. In most cases, this is done by calling a python script named *generate_data.py*, which writes the table contents out as a CSV file, and then importing them using the *psql* utility. Thus, the tables are removed and entirely re-created before each run. Benchmarking-results, which consist of multiple output files for easy analysis, such as the hypergraph, hypertree, a drawing of the hypergraph, and statistics, are saved into a new directory structure analogously to the database directory structure. The *size* of a database is specified by an abstract integer unit, which controls the size of the tables, but has different effects, depending on the database. The reason such an arbitrary value was chosen is for simplicity in the benchmarking process. For a better estimate of query complexity, the output size of the query could also be taken, which is recorded in the benchmarks.

In order to guarantee reliable results and remove the overhead of statistics computation during query execution, all statistics are cleared and computed before each run for both the original and the optimized execution. The database is also fully cleared before each run to ensure that no data other than the currently used tables exist. A practical issue that came up was that result sets were being allocated very quickly while benchmarking, which caused the JVM garbage collector to use up more memory than allowed. Hence, the OS sometimes stopped the process and inaccurate runtime measurements due to garbage-collection pauses occured. To prevent this, the garbage collection process is manually started before benchmarking each query and between benchmarking the optimized and the original query. Otherwise, the amount of memory marked to free might rise during the first execution and be freed just while measuring the execution of the second query (we have observed this effect while benchmarking).

By default, a timeout of 25 seconds is set for the execution of a query, since some queries might take long to run before resulting in an out-of-memory error or possibly even using up the available disk space. Memory consumption and swapping proved to be a major bottleneck on some queries, therefore the Java process is given a generous amount of 45GB RAM by passing the option *-Xmx45G* and the system is equipped with 64GB RAM.

[1] `https://github.com/arselzer/HTQueryOptimizer/tree/04060b28a8805fa2e6eaecd5f73a225aa1643a43/data`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For greater reliability, multiple runs can automatically be performed for each query and size. However, since runtimes turned out to be sufficiently stable, this was only used for validating the benchmarking technique.

Correctness checks are implemented as part of the benchmark. Rows are put into a hash-map for both query executions and the counts are compared. This however takes up a large amount of memory and adds to the runtime, therefore it is also only used for validation of the optimization technique and deactivated during performance-measurement benchmarks.

A special benchmarking mode was implemented to perform only boolean queries. The original queries are automatically modified by appending a **LIMIT** 1 statement at the end and compared to the boolean execution of the optimized query (see Section 5.5.3). This case is of interest since the overhead of the row output and transfer disappears, and boolean queries can be found in real-world situations.

For each benchmarking run, the following data is saved to a CSV file:

- Original query runtime (**origRuntime**)
- Optimized query total runtime (**optTotalRuntime**)
- Optimized query execution time (excluding the time to decompose optimize) (**optQueryRuntime**)
- Number of output rows (**origRows/optRows**). If a timeout occurs in the original or optimized query, the value is set to 0.
- Number of output columns (**origCols/optCols**). This number is expected to be the same assuming the correctness of the optimized query, but it can vary between the original and optimized query in two cases: 1) equal columns are reduced to one by the optimization process 2) a timeout occurs.
- Occurrence of a timeout in the (un)optimized query (**origTimeout/optTimeout**)
- Hypertree height (**treeHeight**)
- Hypertree number of nodes (**treeNodes**)
- (Generalized) hypertree-width (**treeWidth**)
- Hypergraph degree (**hgDegree**)
- Hypergraph VC dimension (**hgVCDimension**)
- Hypergraph BIP (**hgBIP**)
- Hypertree balancedness factor (see Section 2.4) (**balancednessFactor**)
- Vertex bag size (min/max/mean/sum/standard deviation) (**vertexBagSize***)
- Edge bag size (min/max/mean/sum/standard deviation) (**edgeBagSize***)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Hardware

Experiments were carried out on a computer with an *AMD Ryzen 7 PRO 4750U* processor, with 8 cores and 16 virtual cores, 8 MB L3 cache, a base clock speed of 1.7GHz and a maximum single-core clock speed of 4.1GHz. 64GB RAM (DDR4, 3200MHz) is available and 32GB swap space are configured on the 1TB SSD with dm-crypt disk encryption. The OS is Ubuntu with the Linux kernel 5.8.0-41 and PostgreSQL version 12+216.

Data

There are 8 benchmark databases and for each database several queries. They all represent difficult join scenarios with several tables. 6 of the databases were specifically created for this thesis, and 2 are based on the data and schema of the TPC-H benchmark. A more detailed description of the data can be found in Section B, where the SQL queries and their hypergraphs can be seen. We will, in this section, briefly go over the main ideas behind some databases and queries. All columns are of type *integer*, except in the TPC-H databases, where string columns also occur. The abstract unit representing the size of the databases is referred to as *dbSize*. Note that there are, in the current version, redundancies in the data and the naming of queries is inconsistent, which could be improved.

db1 Consists of 11 tables `t1` to `t11`, each with two integer attributes a and b. The values are generated by computing the permutations from 1 up to *dbSize* +1 for tables *t3*-*t11* and *dbSize*+5 permutations for tables *t1* and *t2*. For example, if *dbSize* $= 3$, *t3* contains the values $\{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,3)\}$, and so on.

db2 As **db1**, except that all tables contain *dbSize*+5 permutations.

db3 As **db1**, except that all tables contain *dbSize*+1 permutations.

db4 Contains 5 different tables with varying numbers of attributes and duplicates. The data sizes are hardcoded, hence changing *dbSize* has no effect on this database. For more detail, refer to the generator script[2].

db5 Equivalent to **db1**.

db6 Consists of 18 tables: *c1a, c2a, c3a, ..., c1b, c2b, ..., c5c, c6c*. Each table has 3 attributes: *x, y, z*. The idea behind the structure was to construct queries with large intermediate results and small end results. For all *c*a* and *c*b* tables, attribute z is always 1, while attribute x is sampled randomly by taking one from the set $\{0,1\}$, where the probability of chosing 0 is very small. Attribute y is sampled equivalently, but instead from the set $\{0,2\}$. For the *c*c* tables, z is set to 2, while x and y are samples as previously, but from the sets $\{0,3\}$ and $\{0,4\}$, respectively.

[2] `https://github.com/arselzer/HTQueryOptimizer/blob/42d6f0bcbf2f424001019aff9e1660f467df8e40/data/db4/create.sh`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

fat_and_cycles Consists of several tables: *fat(a,b,c,d)*, *bad(xbad, ybad)*, *good(ugood, wgood)*, *u(c, ugood)*, *w(d,wgood)*, *x(a,xbad)*, and *y(b,ybad)*. These tables are then naturally joined.

tpc-h Contains the TPC-H tables generated with the command dbgen –s 0.15, where the parameter *s* refers to the scale factor of the database.

tpc-h-2 Contains the TPC-H tables generated with the command dbgen –s 0.025.

6.1.2 Configuration

The following commands were used to generate the results presented here:

All results can be downloaded from the GitHub release: `https://github.com/arselzer/HTQueryOptimizer/releases/tag/final-data`. The configuration used, as well as the script to generate the diagrams (*visualize.ipynb*), is found in the git tree of the commit mentioned in Section 6.1.

1. `java -Xmx45G -cp "target/HTQueryOptimizer-1.0-SNAPSHOT.jar" "benchmark.Benchmark" -timeout 100 -parallel -boolean`
2. `java -Xmx45G -cp "target/HTQueryOptimizer-1.0-SNAPSHOT.jar" "benchmark.Benchmark" -timeout 100 -parallel`
3. `java -Xmx45G -cp "target/HTQueryOptimizer-1.0-SNAPSHOT.jar" "benchmark.Benchmark" -timeout 100`
4. `java -Xmx45G -cp "target/HTQueryOptimizer-1.0-SNAPSHOT.jar" "benchmark.Benchmark" -timeout 100 -parallel -unweighted`
5. `java -Xmx45G -cp "target/HTQueryOptimizer-1.0-SNAPSHOT.jar" "benchmark.Benchmark" -timeout 100 -parallel -threads 1`
6. `java -Xmx45G -cp "target/HTQueryOptimizer-1.0-SNAPSHOT.jar" "benchmark.Benchmark" -timeout 100 -parallel -threads 2`
7. the previous setting with 3, 4, ... 8 threads

First, boolean and non-boolean queries are executed with the parallel query executor. Then non-boolean queries are executed using the function-based optimizer. Then, the parallel approach is executed without statistics. Finally, the parallel execution strategy is performed with only 1 (i.e. sequentially), 2 and 3 threads.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2 Overview of Results

We will now present the results of the experimental evaluation. First, the results of PostgreSQL against the new optimizer on the answer enumeration problem are given, then the results on the semi-join versions of the same queries. Next, we present data to investigate the effect of statistics. Finally, parallel execution in comparison to sequential execution inside a PL/pgSQL function and the effect of the number of parallel threads is presented. After presenting the data, we analyze and discuss the results.

6.2.1 Optimized vs. Non-Optimized Execution

We can observe that, in most cases, PostgreSQL still beats the decomposition-based optimizer by significant amounts. On the TPC-H instances, PostgreSQL has the greatest advantage over our system, especially in the case of query 5. We assume that this is due to the I/O overhead of the temporary tables, in combination with the simple structure of the query (it contains only one cycle), where our system does not provide such a great advantage as on more complex queries. However, its runtime is in most cases not significantly worse than PostgreSQL's and sufficient for terminating in reasonable time in real-world use-cases. On some queries, our system manages to outperform PostgreSQL.

On *db5/query2.sql*, depending on the size of the DB, the decomposition-optimizer performs better than PostgreSQL. The two different runtime behaviours are caused by two different decomposition structures, which are chosen depending on the statistics of the data in the DB.

On one query, *db6/query1.sql*, we found that the system outperforms PostgreSQL significantly. This can be explained by the large intermediate results produced in the query although the end result is very small. In such a situation, Yannakakis' algorithm is very effective due to the reduction of intermediate results by semi-joins.

6.2.2 Performance on Boolean Queries

In the case of boolean queries, our system is significantly more competitive than on the enumeration of answer sets. Although it can be seen that there is an overhead on fast-to-answer queries (e.g., *db1/triangle.sql*), it proved quite effective on more complex queries. The strongest improvement can be observed on query *db4/query-5-empty.sql*, a query with potentially large intermediate results, but no end results, where the original query takes over 30 seconds to run but the optimized version runs in 181 milliseconds (238 including optimization). The weak point of our system are again the TPC-H queries, where PostgreSQL runs query *tpc-h/5-modified.sql* in 40 ms while our system takes 2406 to answer it.

The performance improvements on boolean queries can be explained by the full-join step as well as the top-down semi-join step falling away, where the largest overhead of our system is found.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2.3 The Effect of applying Weighted Decompositions

On most simpler queries, the performance of weighted and unweighted decomposition-based query execution is approximately the same (e.g., all queries of *db1, db2, db3*, and *db4*). In more complex cases, weighted decompositions become a great advantage, or even the only way for the query to execute before reaching a timeout or using up the resources of the system.

Query *db5/query2.sql* does not terminate in most cases using unweighted decompositions. In the case of *db6/query1.sql* and *fat_and_cycles/query-no-star.sql*, the use of weighted hypergraphs leads to a speedup by a factor of 10. The bad performance of unweighted hypergraphs can frequently be explained by the occurrence of cross-products in the decompositions, while the weighted approach effectively avoids these as well as expensive joins.

6.2.4 Parallel Execution vs. PL/pgSQL Execution

We can observe a slight trade-off between the parallel execution of optimized queries and PL/pgSQL function-based execution. On queries of lower runtime, such as *db1/triangle-star.sql*, the PL/pgSQL-based execution is superior due to less communication overhead caused by repeatedly executing SQL statements. Instead, the execution is performed fully inside the DBMS. On more complex queries of greater runtime, such as *db6/query1.sql*, the parallel execution approach tends to be the faster one, since the parallelization has a positive effect on the runtime when multiple long-running SQL statements are executed in parallel, which offsets the communication overhead.

6.2.5 Parallel vs. Sequential Execution

We also compared the performance of the parallel query executor while varying the number of threads. By setting the number of threads to 1, it can effectively be turned into a sequential execution outside of a PL/pgSQL function. We measured the runtimes with thread counts between 1 and 8. The result was that parallelization does have a positive effect on many queries, sometimes a significant one. On *db6/query1.sql*, for example, the runtime is reduced from 155 ms with one thread to 75 ms with 8 threads, as shown in Figure 6.2. The greatest improvement in performance is achieved between sequential execution and the use of 2 threads. Using more than 4 threads did not result in great improvements on the data sets considered in this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

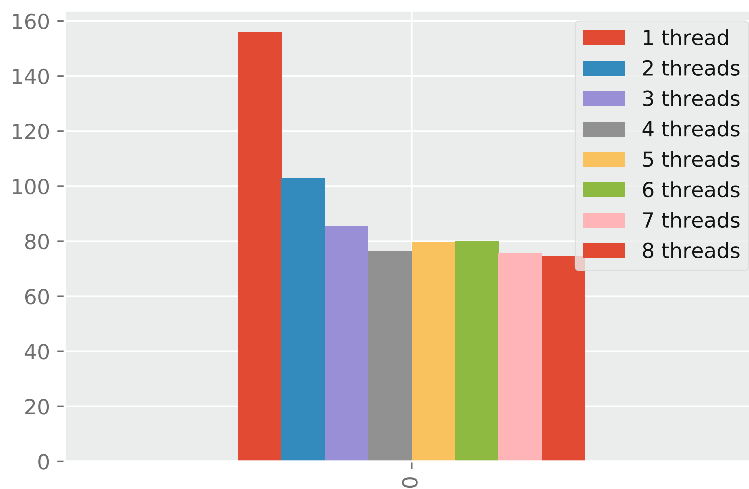


Figure 6.2: The average performance over all DB sizes on *db6/query1.sql* depending on the number of threads

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Conclusion

We have shown that a lightweight integration of structural decomposition-based query-optimization techniques into a DBMS is feasible, and can even be an effective approach towards query optimization, without a full integration. By benchmarking PostgreSQL against the newly-developed system, we verified that the approach is competitive and identified cases where it is superior to the standard optimizer.

On many boolean queries, a class of queries corresponding to SQL `EXISTS` statements over joins, the system proved to be highly effective, even outperforming PostgreSQL significantly.

The integration of statistics and the use of weighted decompositions proved essential for the competitiveness of the decomposition-based optimizer. Our results show that not only the width of the decomposition, but also the structure of it with respect to the query and database, needs to be considered. If the decomposition is only optimized for width, it will in many cases lead to sub-optimal query executions.

Query execution based on Yannakakis' algorithm can successfully make use of parallelization. We have shown that it is possible to achieve a speedup through the parallelization of query execution along the decomposition tree.

7.1 Future Work

Evaluating the optimizer with a more extensive collection of databases and queries than the constructed queries and TPC-H, preferably on more real-world queries would allow identifying the strengths and weaknesses of the implementation more effectively. The case of complex boolean queries with potentially empty result sets appears especially worthy of further investigation. Furthermore, parallelizable queries, such as *db6/query1.sql*, where a large amount of the computational effort comes from the joins inside the bags might be interesting to investigate.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As explained in Section 5.3, our implementation of weighted decompositions currently only considers the sum of the bag-weights. Including semi-join costs as well would be expected to have a positive impact on query performance. A possible further approach to improve the system is the reliable identification of the optimal execution parameters, such as PL/pgSQL execution vs. parallel execution and the number of threads.

The current system is limited to select-from-where queries of restricted syntax. An extension to the full SQL syntax, such as aggregations and subqueries would be necessary for practical use of the system of all possible join queries.

An integration of a (G)HD-based query-optimizer into the core of PostgreSQL or other database systems is still of great interest, since the overhead of the external implementation would fall away. As a further step integration of worst-case optimal join algorithms into database systems is a potentially very promising approach for future research, but would require significant changes to the core of database systems.

The result that queries can be intercepted before entering the DBMS and broken up into parts to run in parallel, with a runtime improvement, is interesting by itself. It would be interesting to investigate this approach for increasing scalability in database systems.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

APPENDIX A

Benchmark Results

In this part of the Appendix, the runtime measurements from the benchmarks are presented. Wherever runtimes are missing due to timeouts, no bar is displayed. The runtimes shown in the following graphics always refer to the execution runtime, excluding the optimization overhead (for the full data, refer to the results linked in Section 6.1.2). The bar chart labels refer to the database sizes in Sections A.1 and A.2, and to *database / query / size* in Sections A.3, A.4, and A.5.

For a description of the benchmark data, refer to Appendix B.

A.1 Optimized vs. Non-Optimized Performance

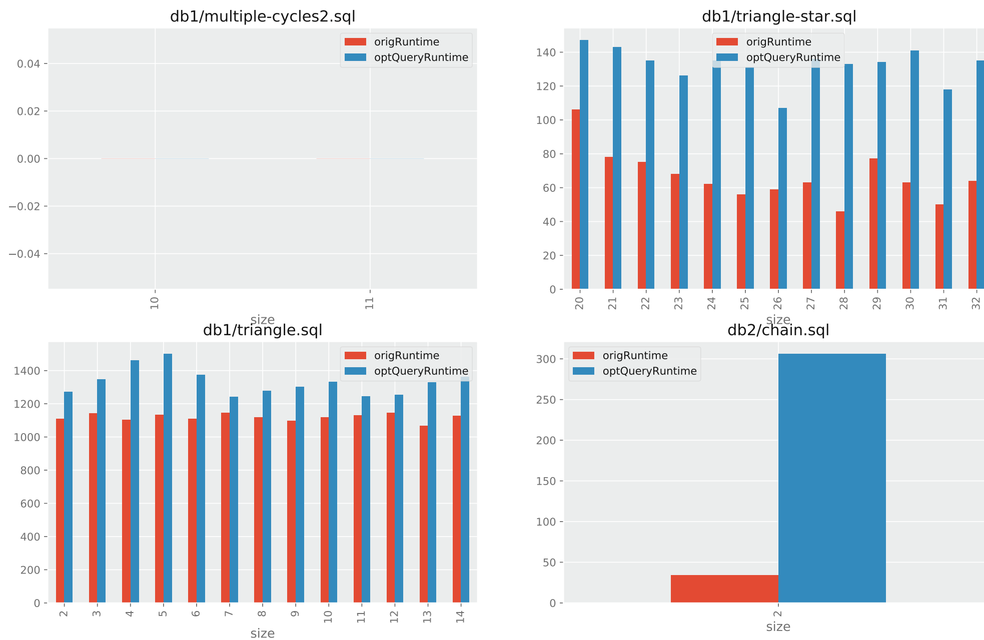


Figure A.1

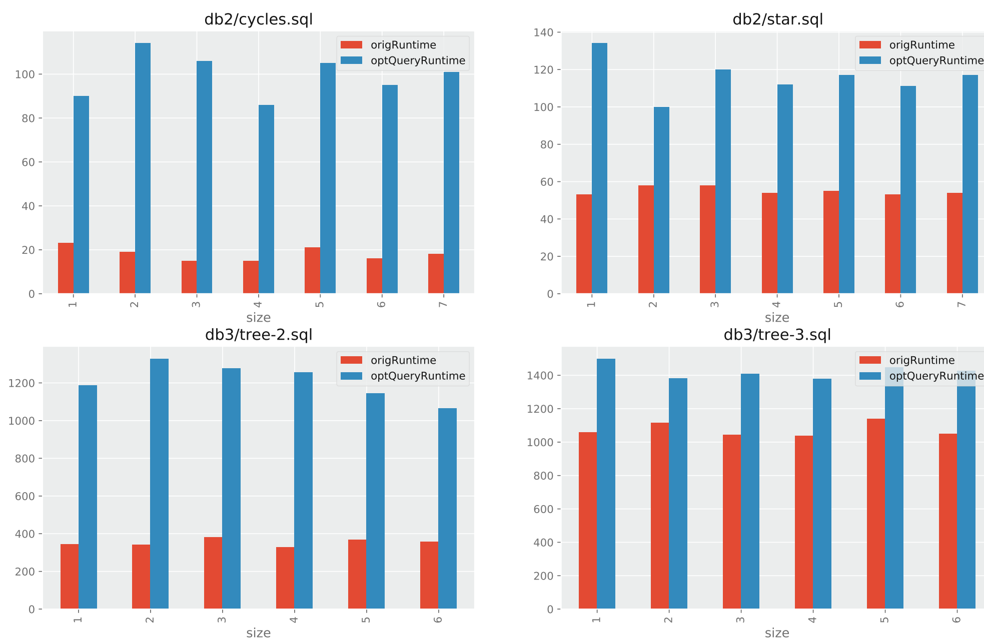


Figure A.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

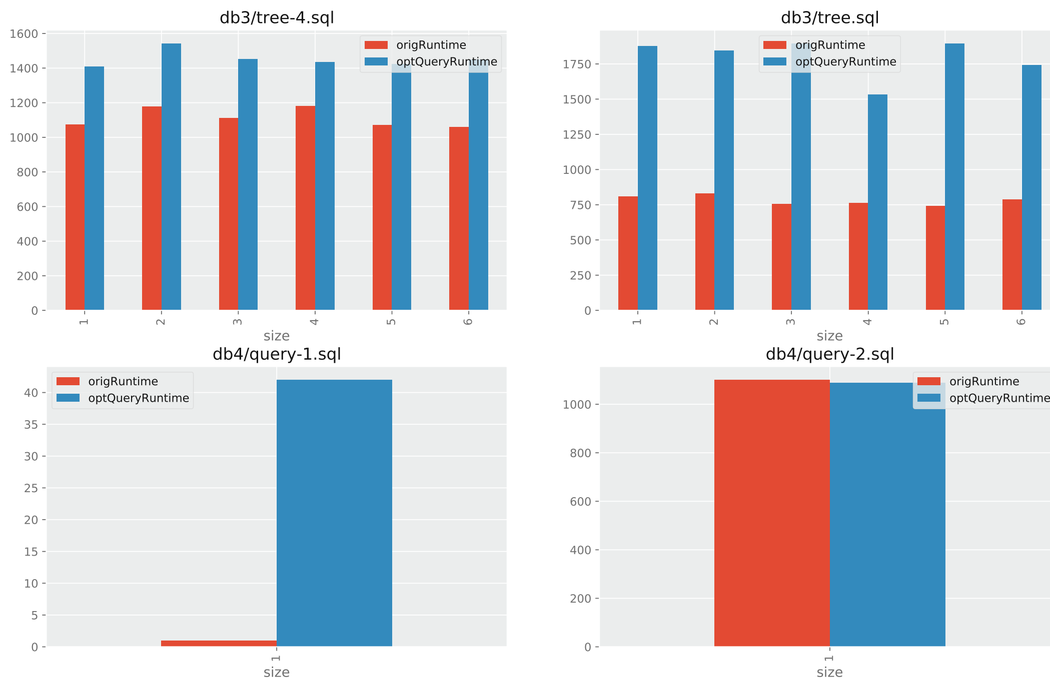


Figure A.3

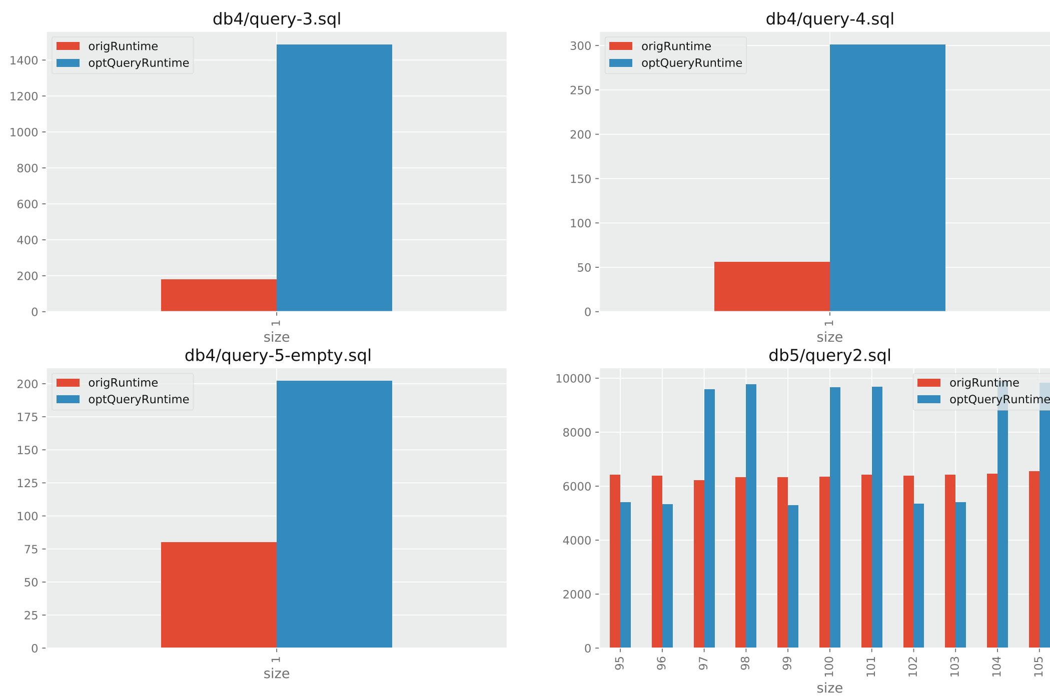


Figure A.4

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
Bibliothek, Your knowledge hub
TU WIEN

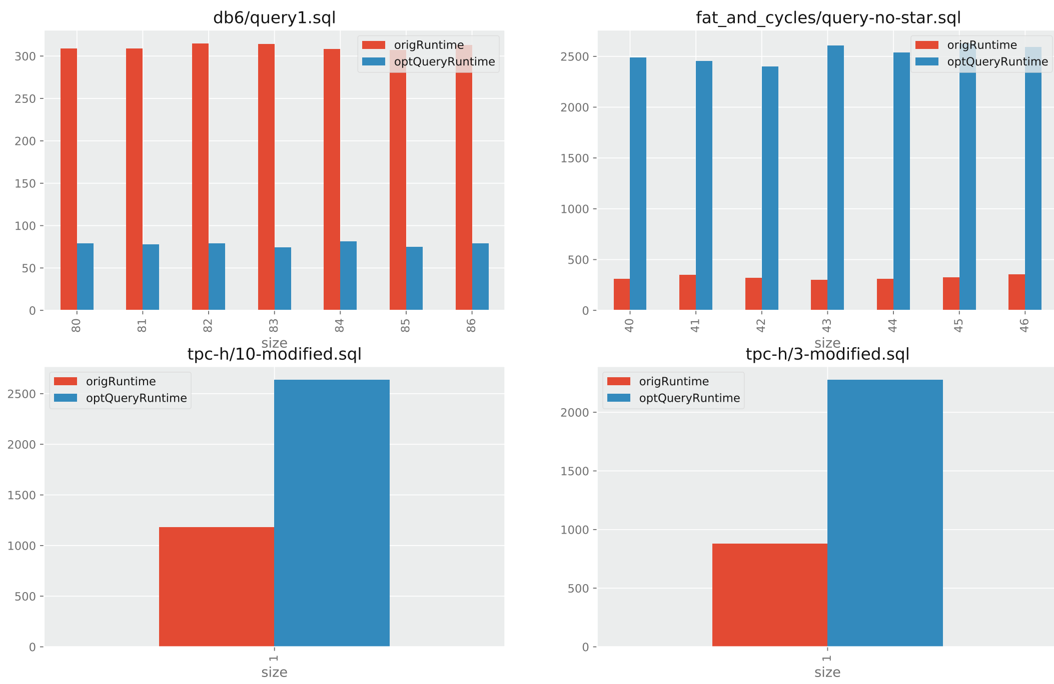


Figure A.5

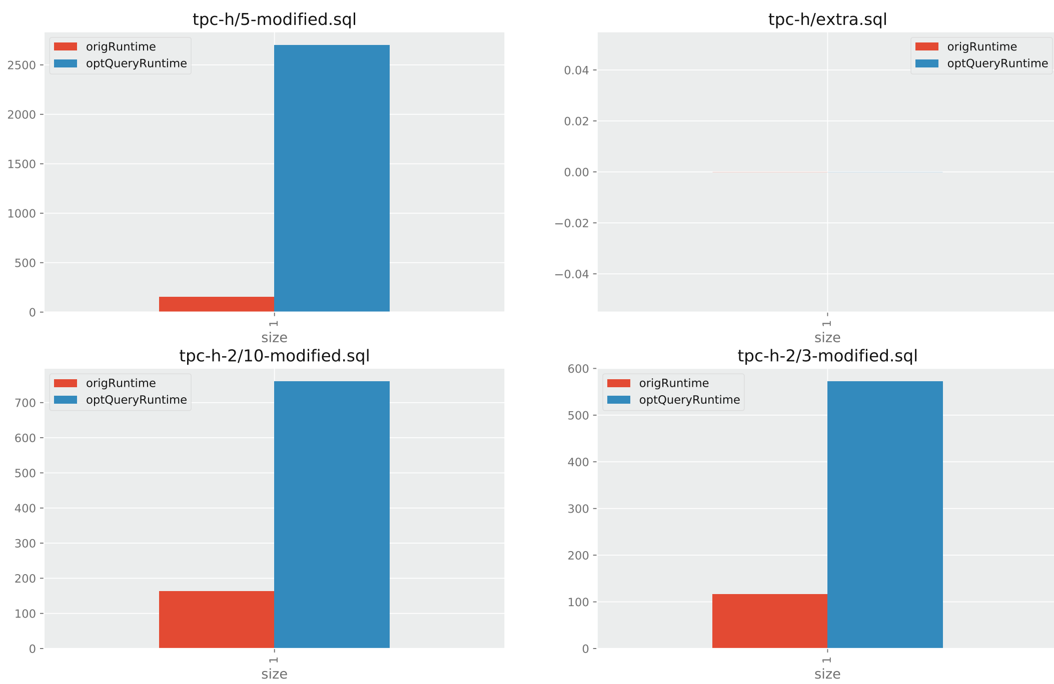


Figure A.6

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

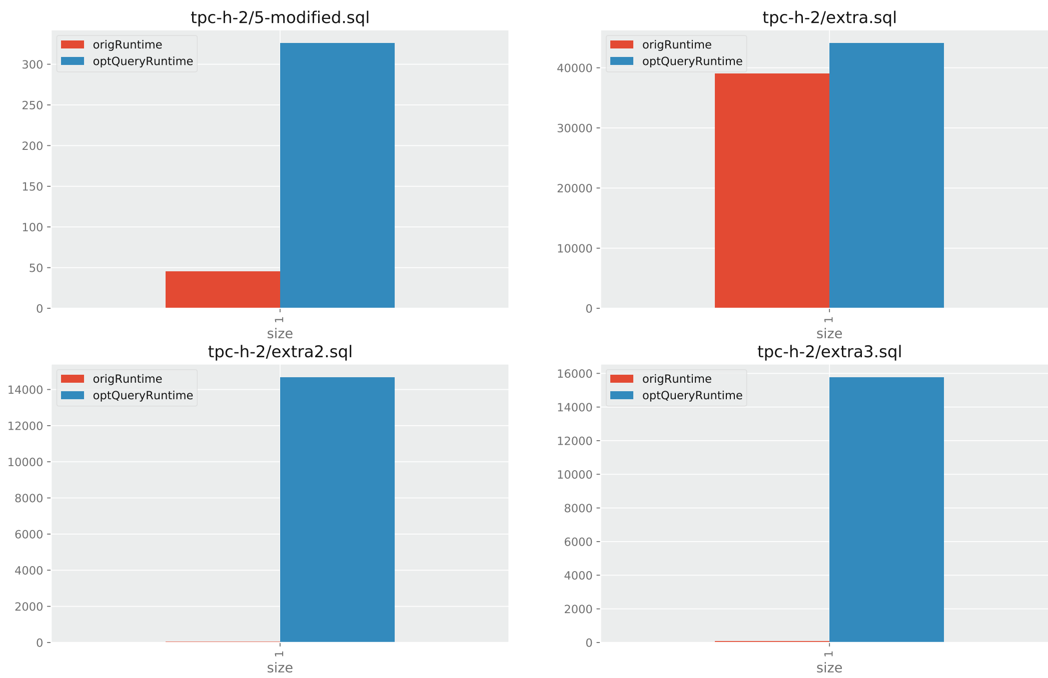


Figure A.7

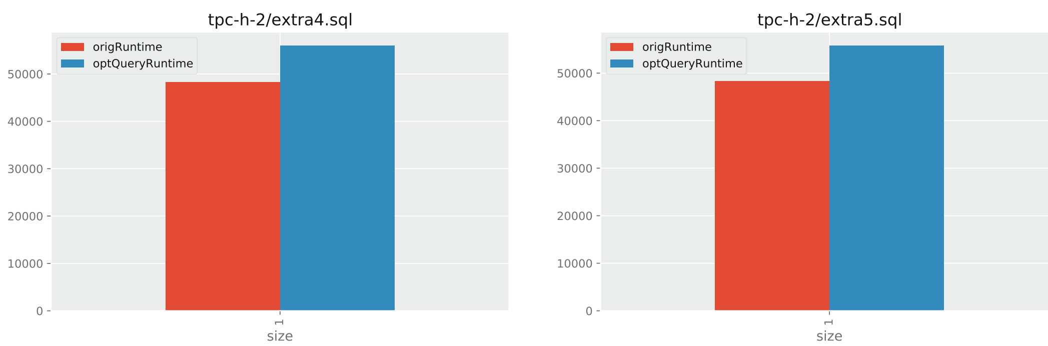


Figure A.8

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A.2 Performance on Boolean Queries

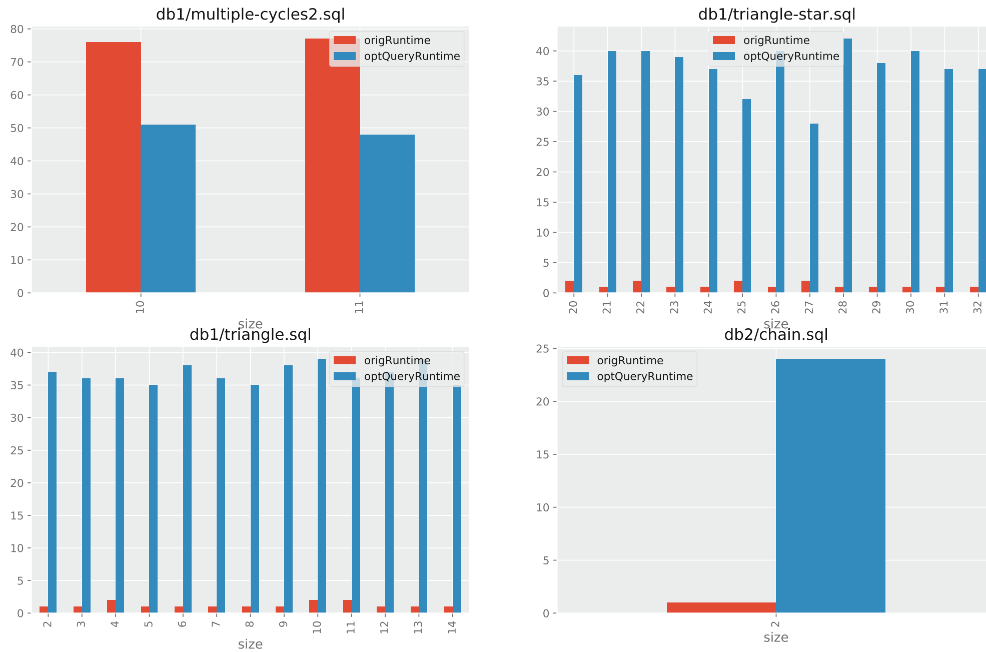


Figure A.9

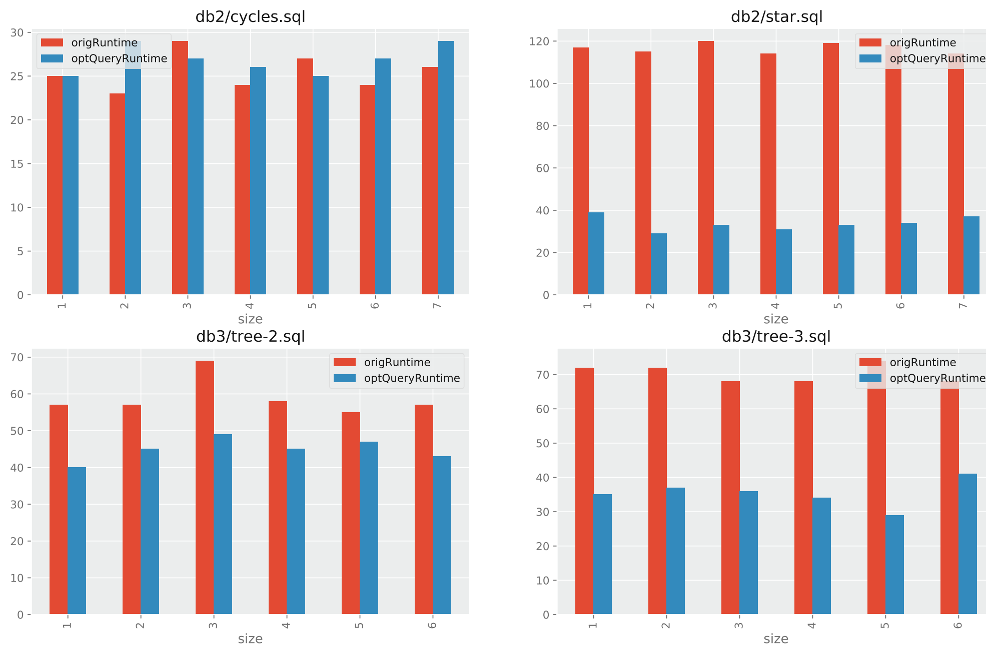


Figure A.10

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

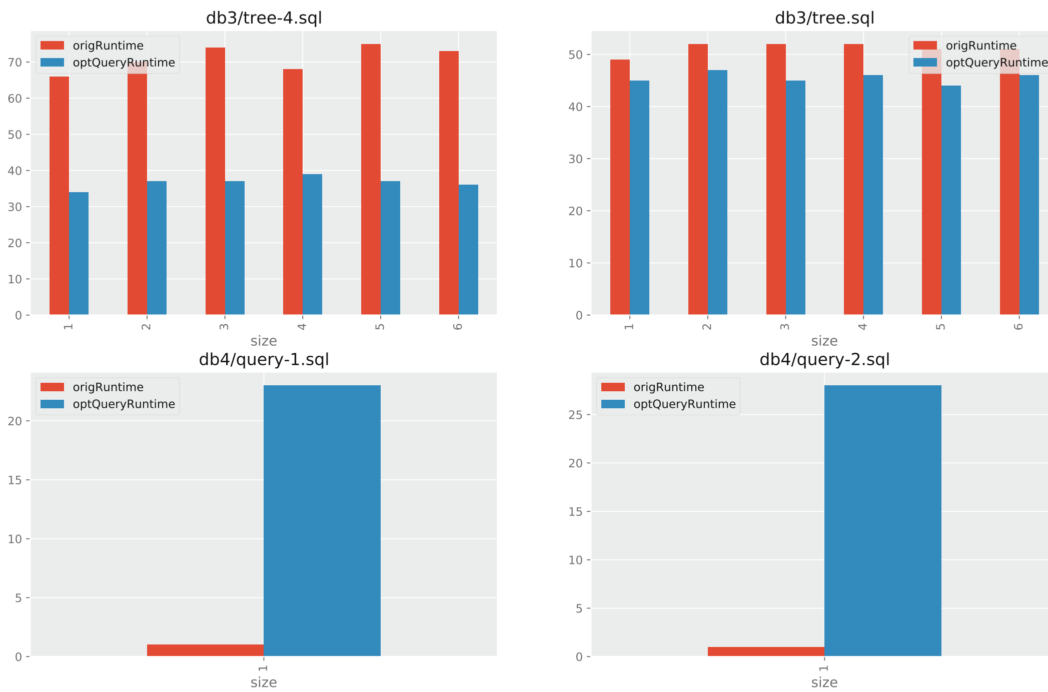


Figure A.11

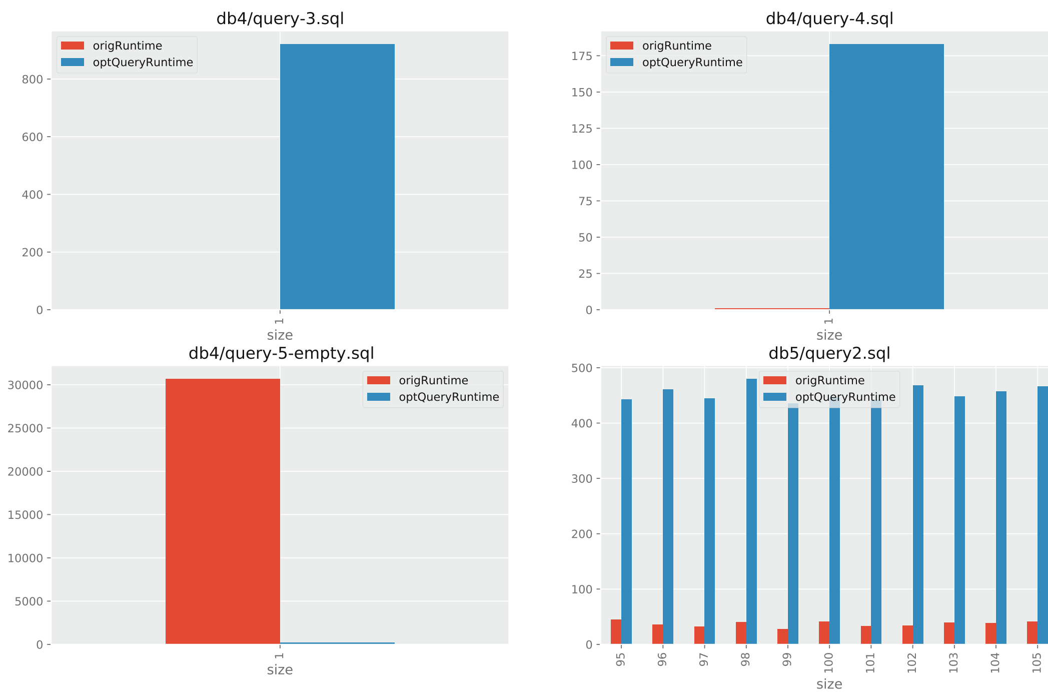


Figure A.12

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

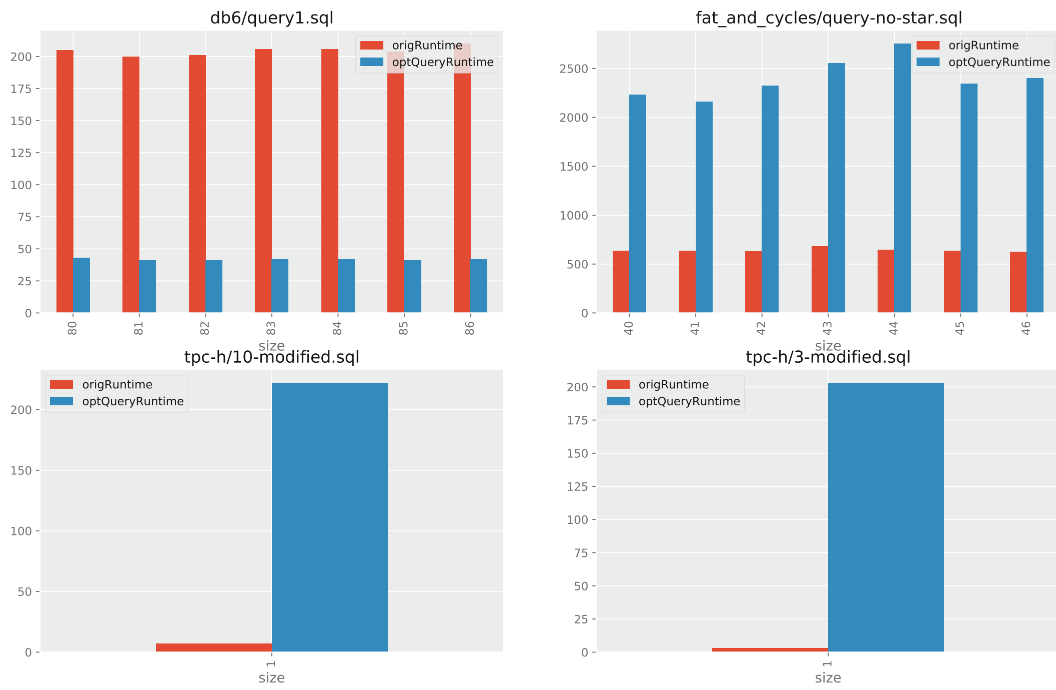


Figure A.13

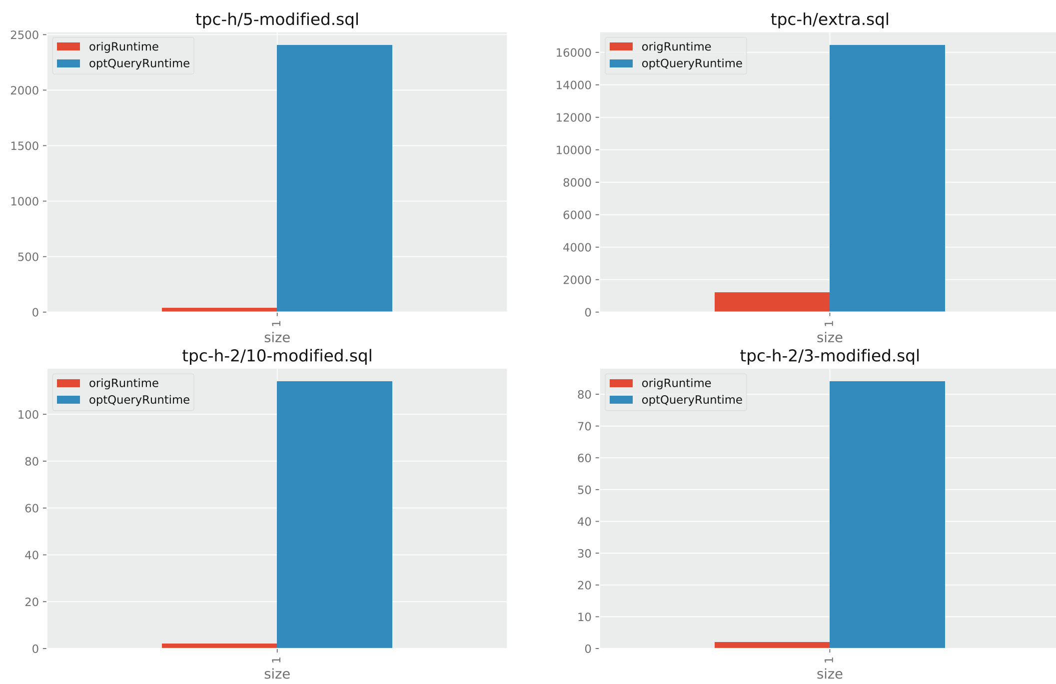


Figure A.14

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

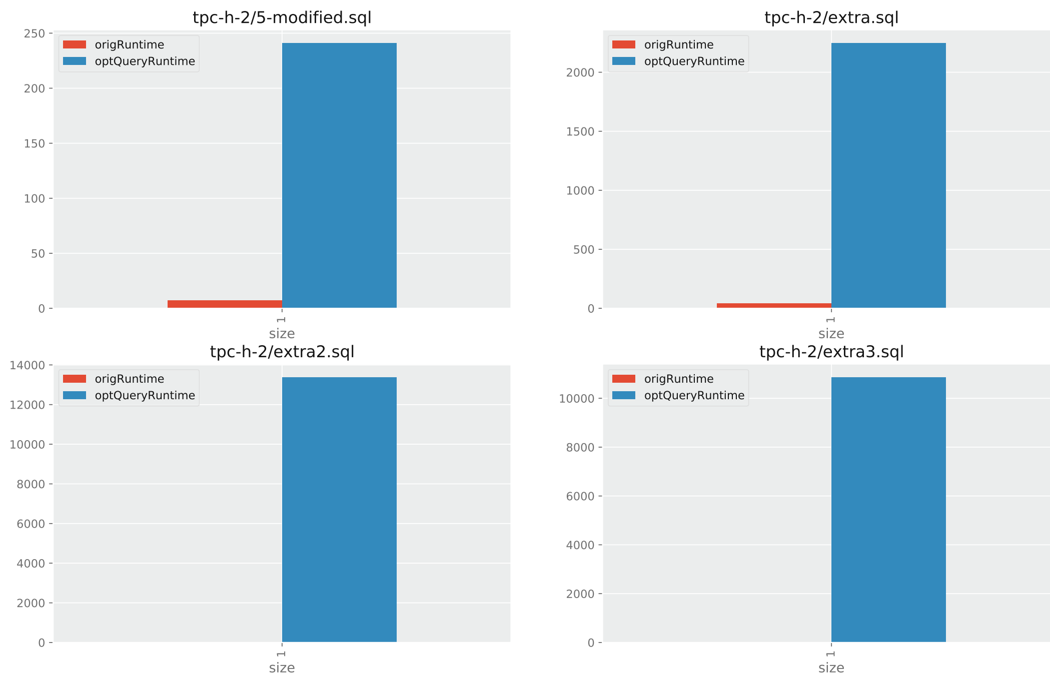


Figure A.15

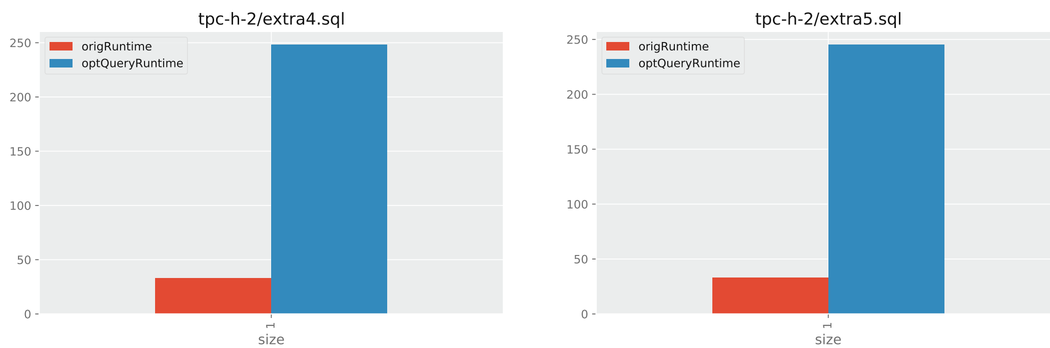


Figure A.16

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
Bibliothek Your knowledge hub
TU WIEN

A.3 The Effect of applying Weighted Decompositions

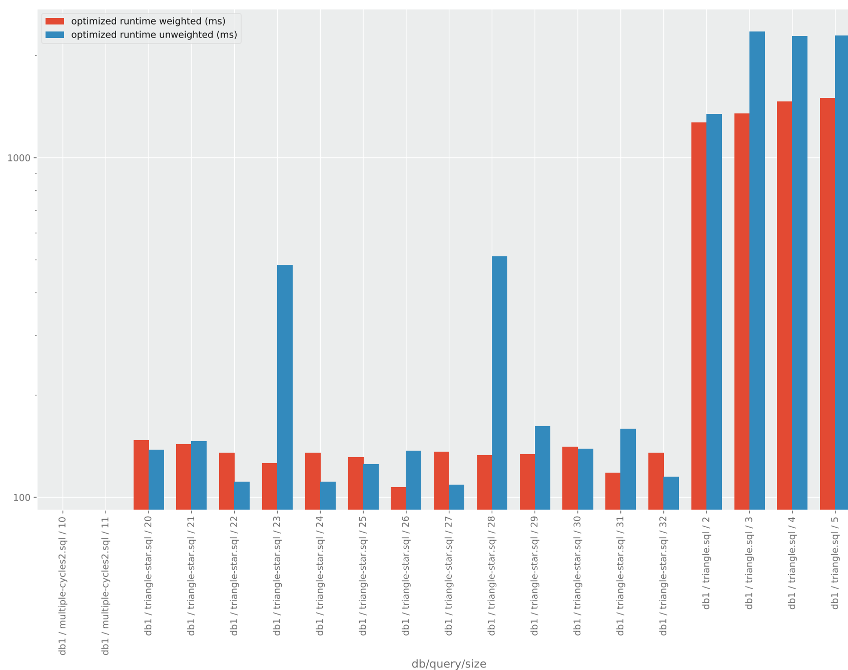


Figure A.17

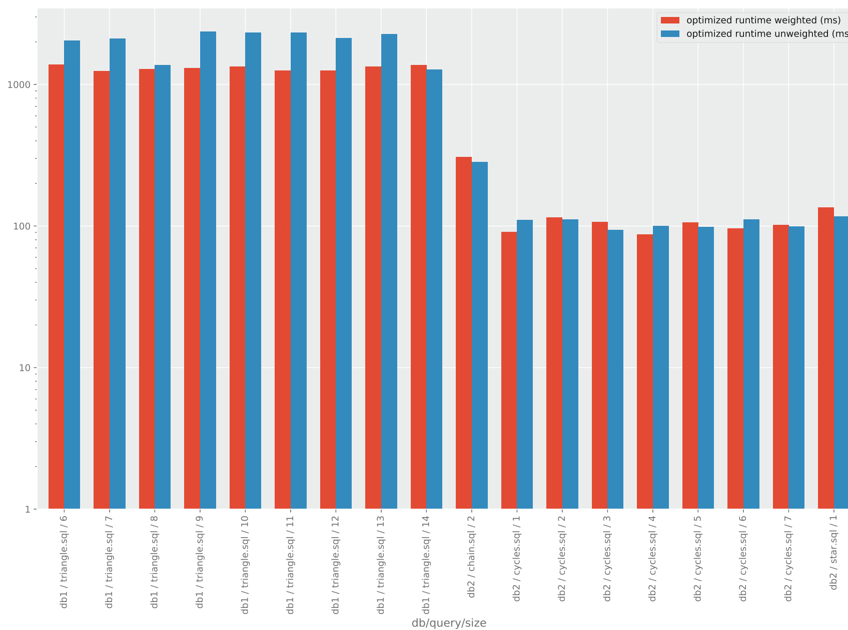


Figure A.18

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

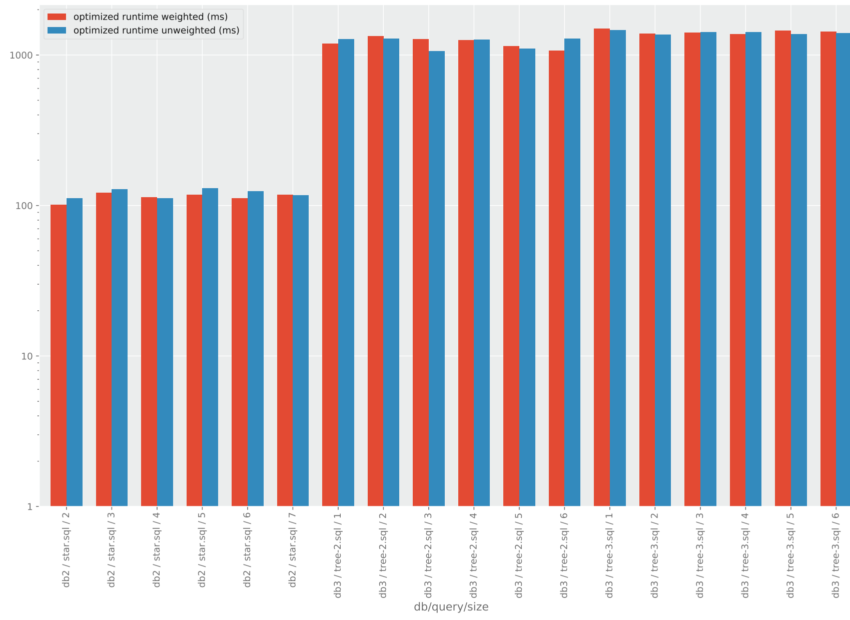


Figure A.19

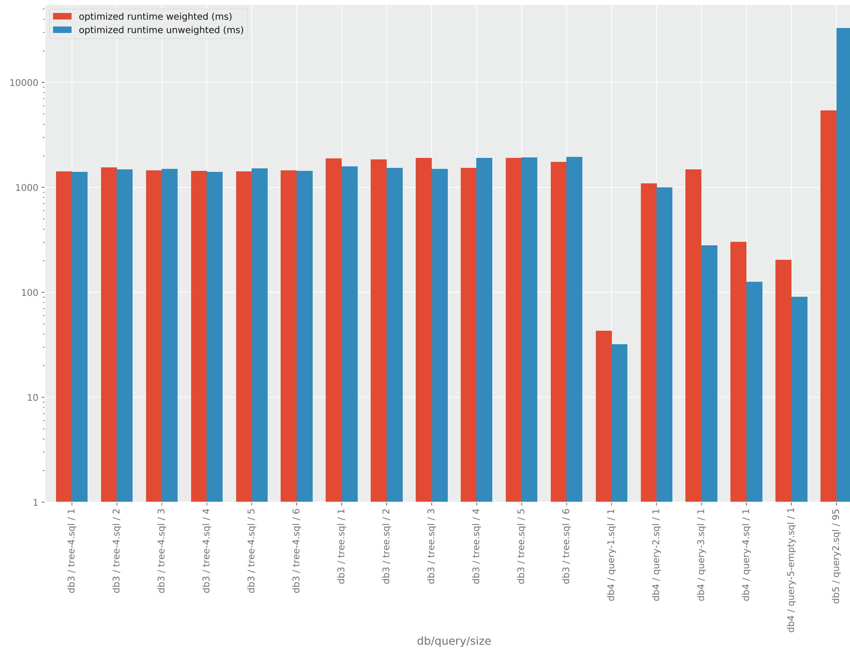


Figure A.20

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

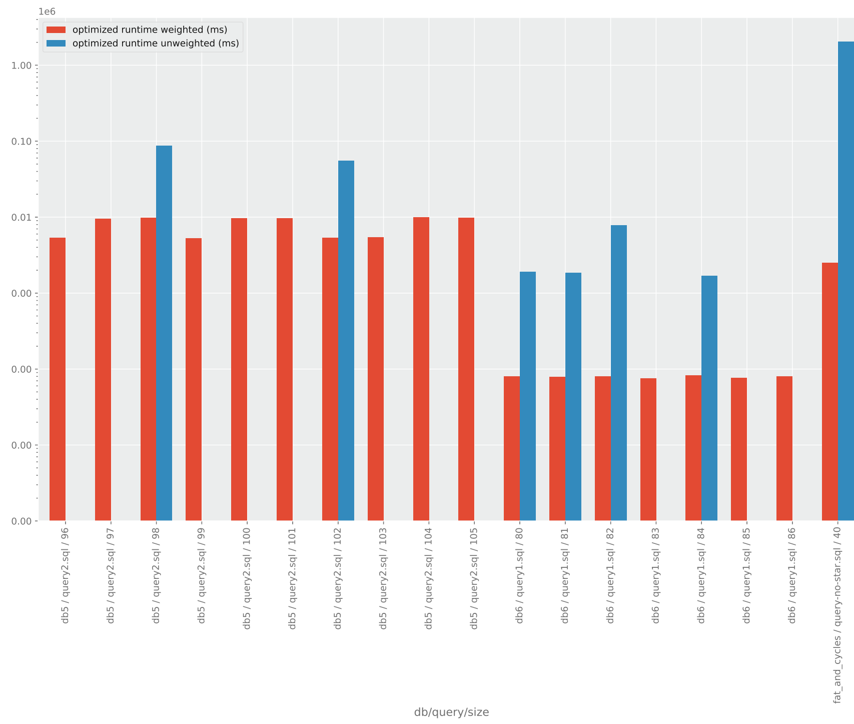


Figure A.21

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

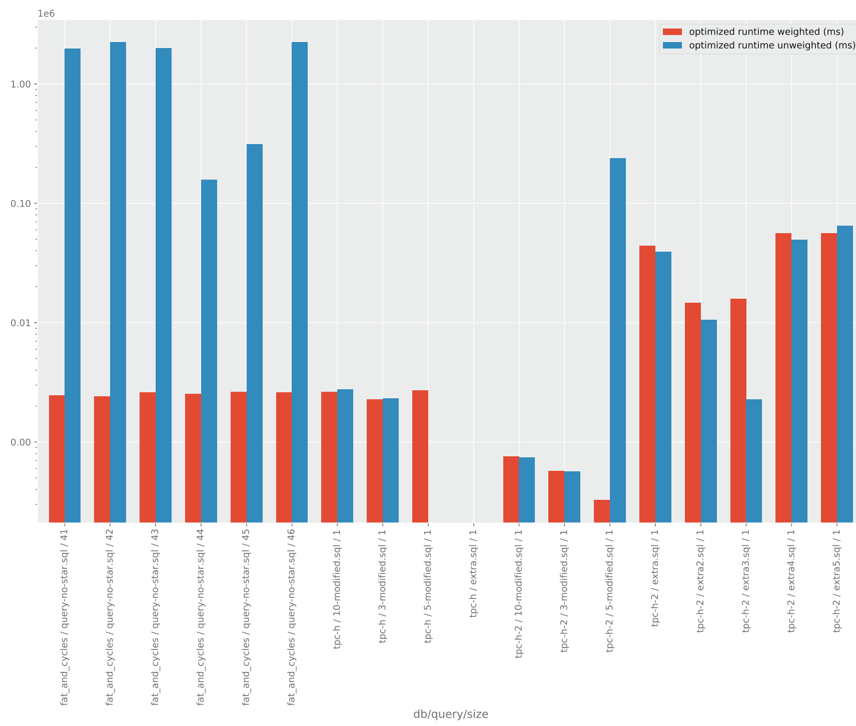


Figure A.22

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A.4 Parallel Execution vs. PL/pgSQL Execution

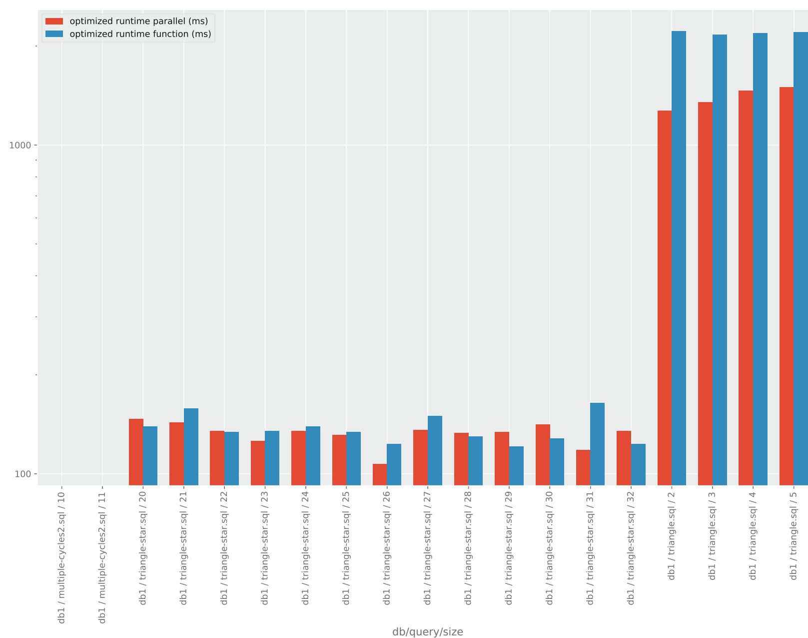


Figure A.23

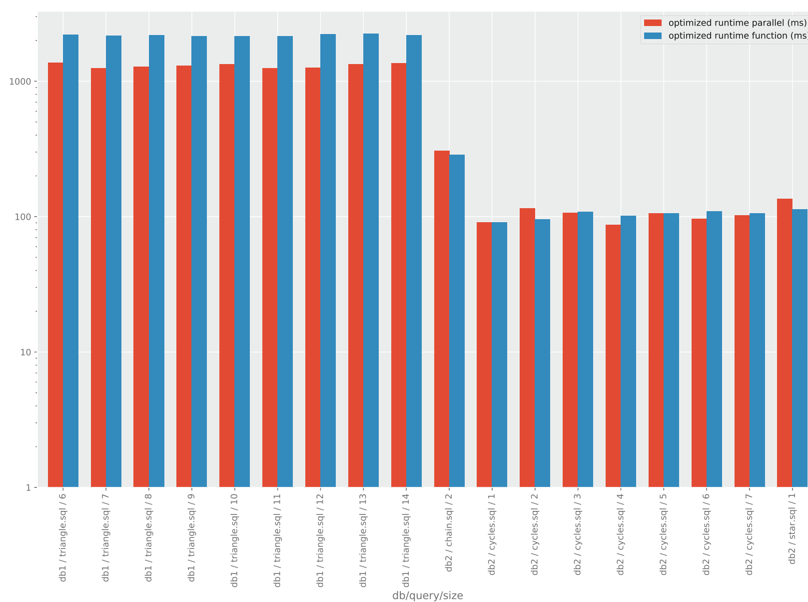


Figure A.24

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

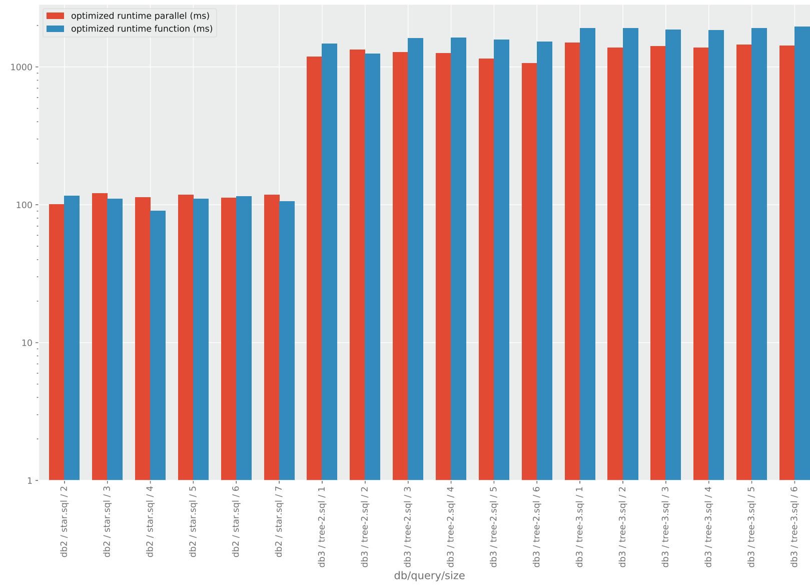


Figure A.25

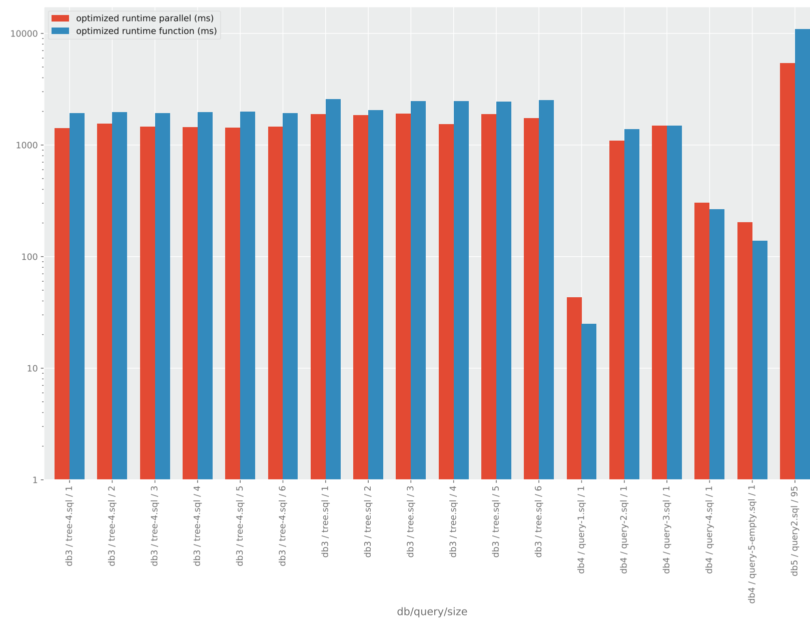


Figure A.26

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

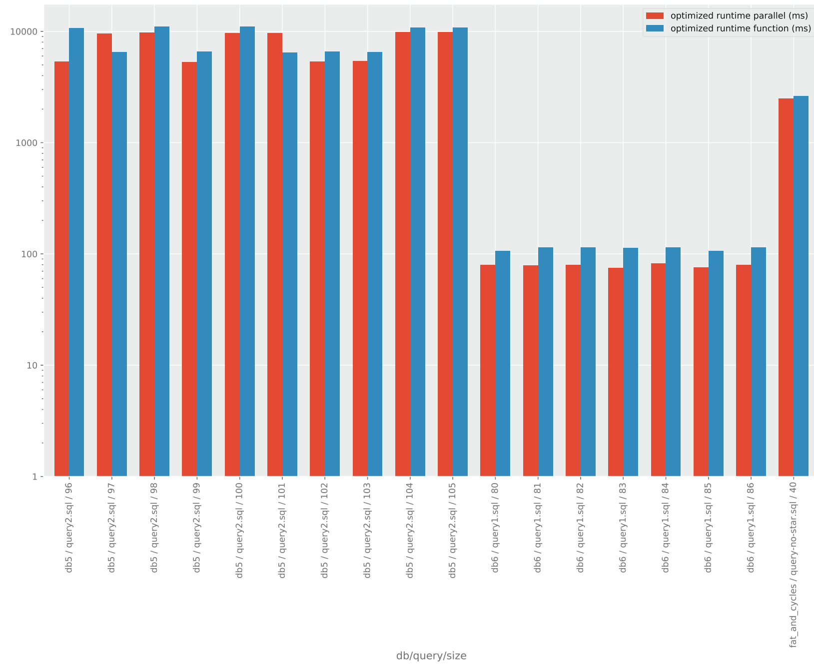


Figure A.27

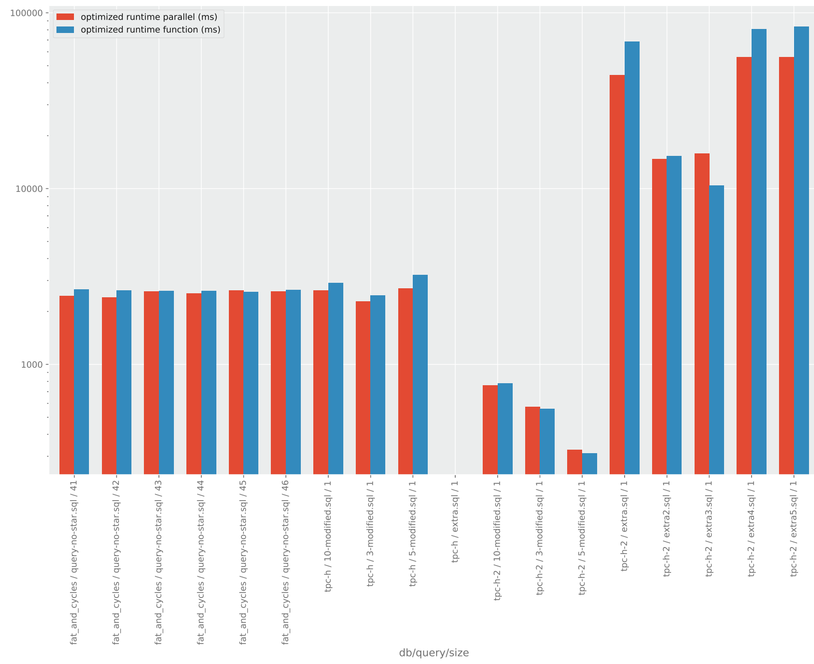


Figure A.28

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A.5 Parallel vs. Sequential Execution

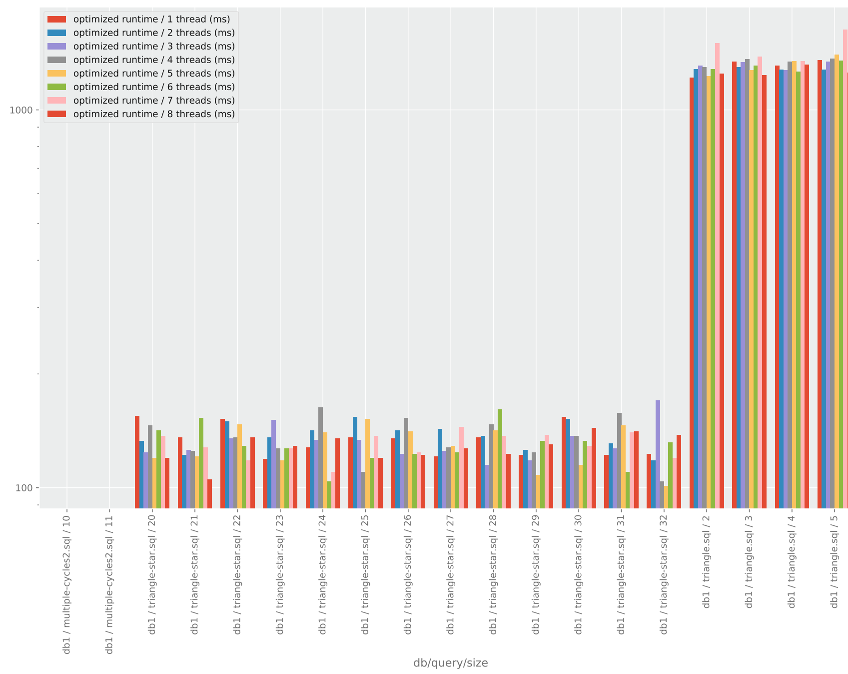


Figure A.29

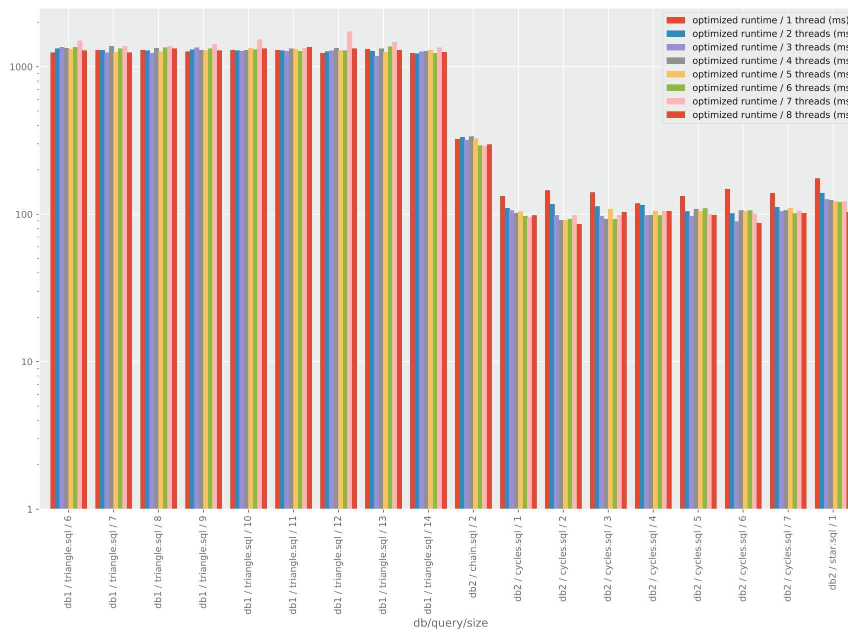


Figure A.30

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

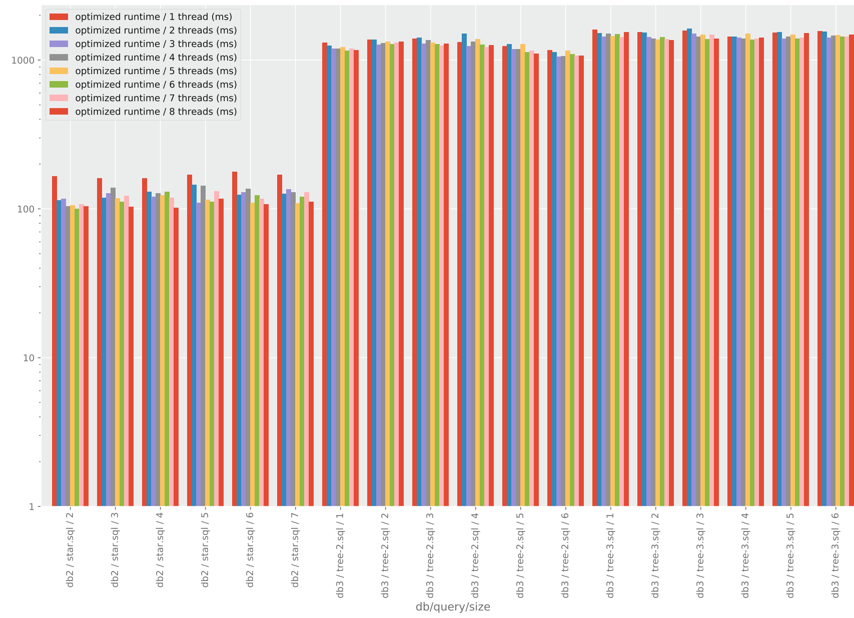


Figure A.31

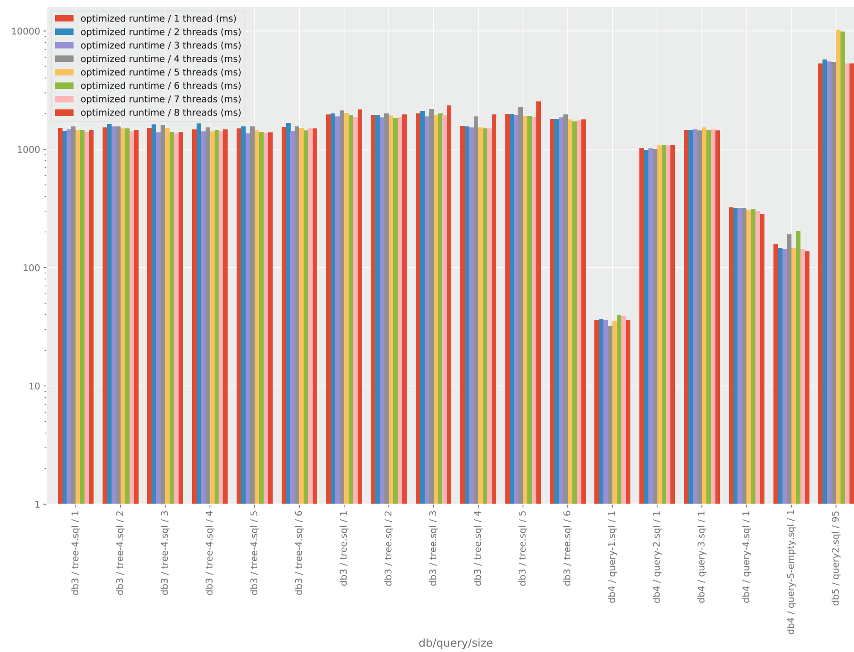


Figure A.32

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

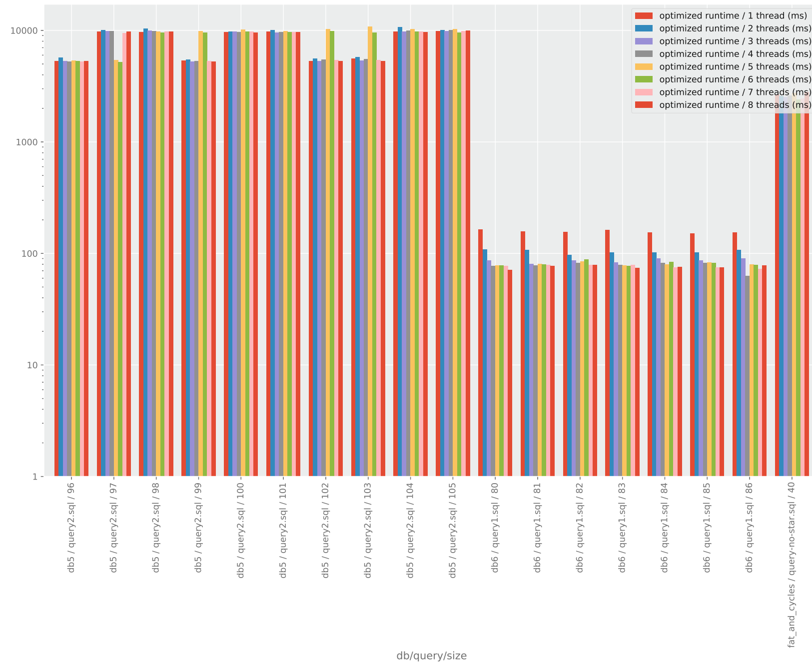


Figure A.33

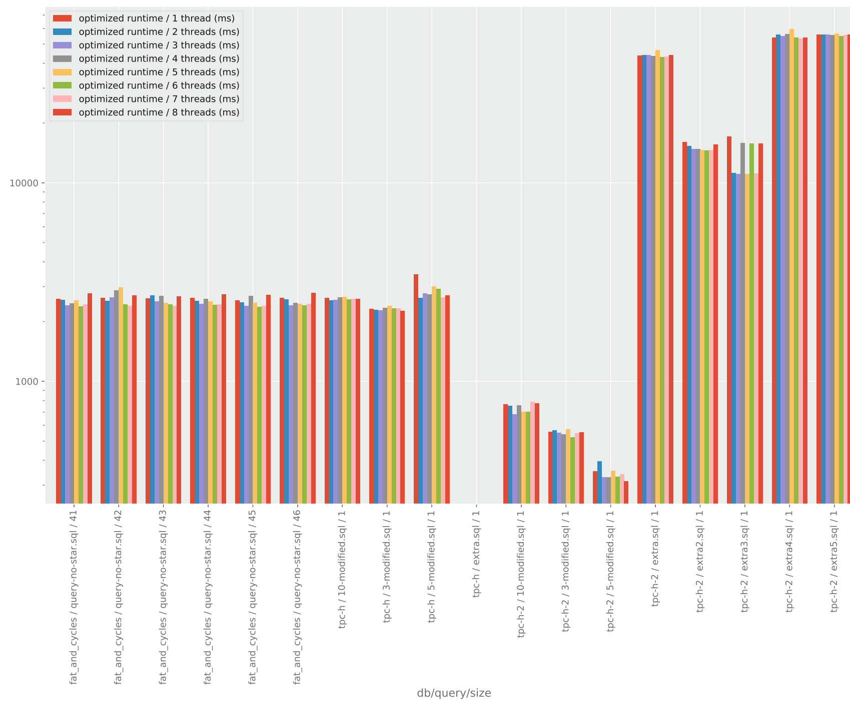


Figure A.34

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX B

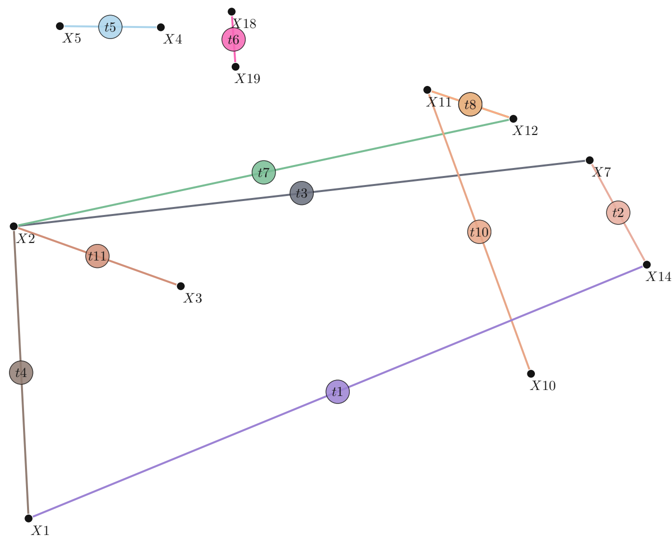
Benchmarking Data

B.1 db1

B.1.1 multiple-cycles2.sql

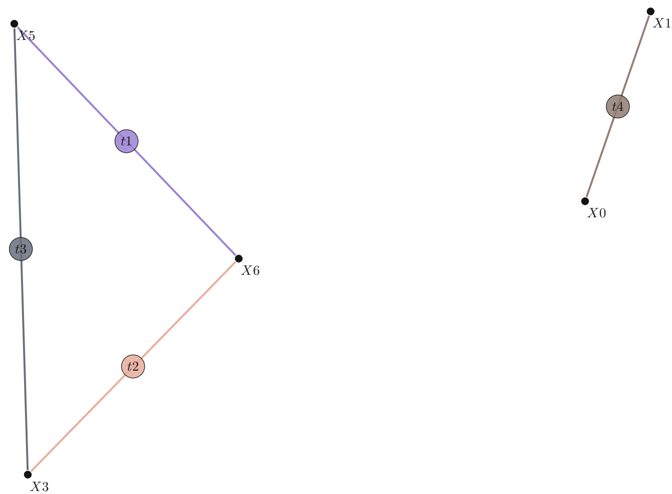
```
SELECT t1.a,t5.b,t10.b
FROM t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11
WHERE t1.a = t2.a
AND t2.b = t3.a
AND t3.b = t4.a
AND t4.b = t1.b
AND t4.a = t7.a
AND t7.b = t8.a
AND t8.a = t9.b
AND t9.a = t10.b
AND t10.b = t8.b
AND t11.a = t4.a;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



B.1.2 triangle.sql

```
SELECT t1.a,t3.b
FROM t1, t2, t3, t4
WHERE t1.a = t2.a
AND t2.b = t3.a
AND t3.b = t1.b;
```

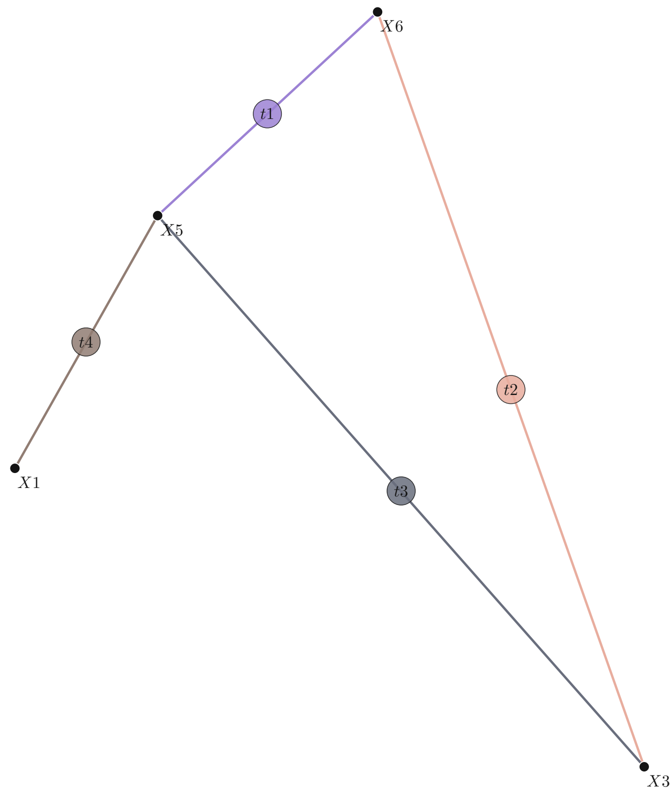


B.1.3 triangle-star.sql

```
SELECT t1.a,t3.a
FROM t1, t2, t3, t4
WHERE t1.a = t2.a
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
Bibliothek, Your knowledge hub
TU WIEN

```
AND t2.b = t3.a
AND t3.b = t1.b
AND t3.b = t4.a;
```

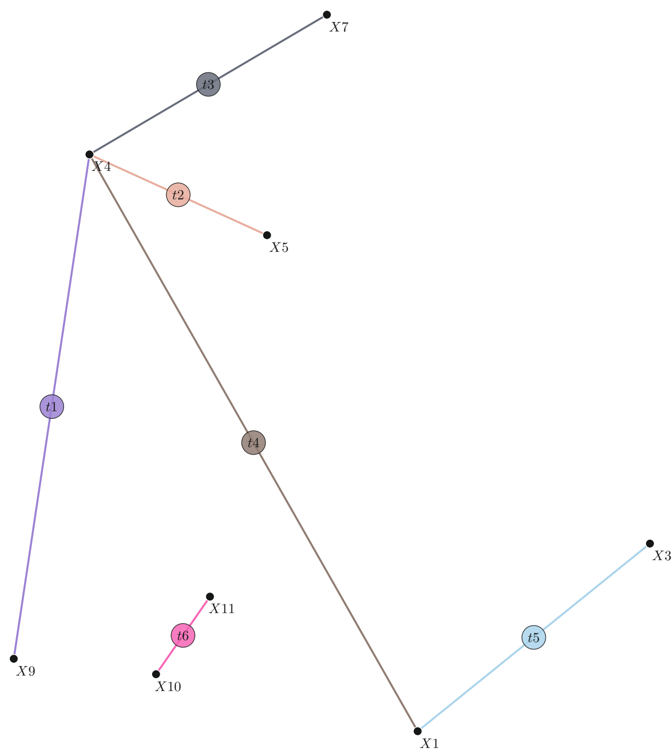


B.2 db2

B.2.1 chain.sql

```
select t1.a, t1.b, t3.b, t5.b
from t1, t2, t3, t4, t5, t6
where t1.a = t2.a
and t2.a = t3.a
and t3.a = t4.a
AND t4.b = t5.a;
```

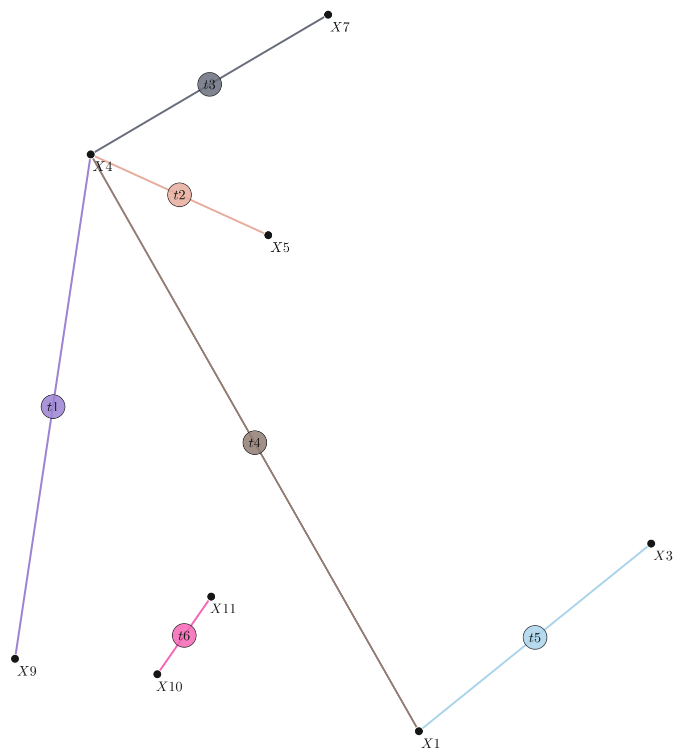

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



B.2.2 chain.sql

```
select t1.a, t1.b, t3.b, t5.b
from t1, t2, t3, t4, t5, t6
where t1.a = t2.a
and t2.a = t3.a
and t3.a = t4.a
AND t4.b = t5.a;
```

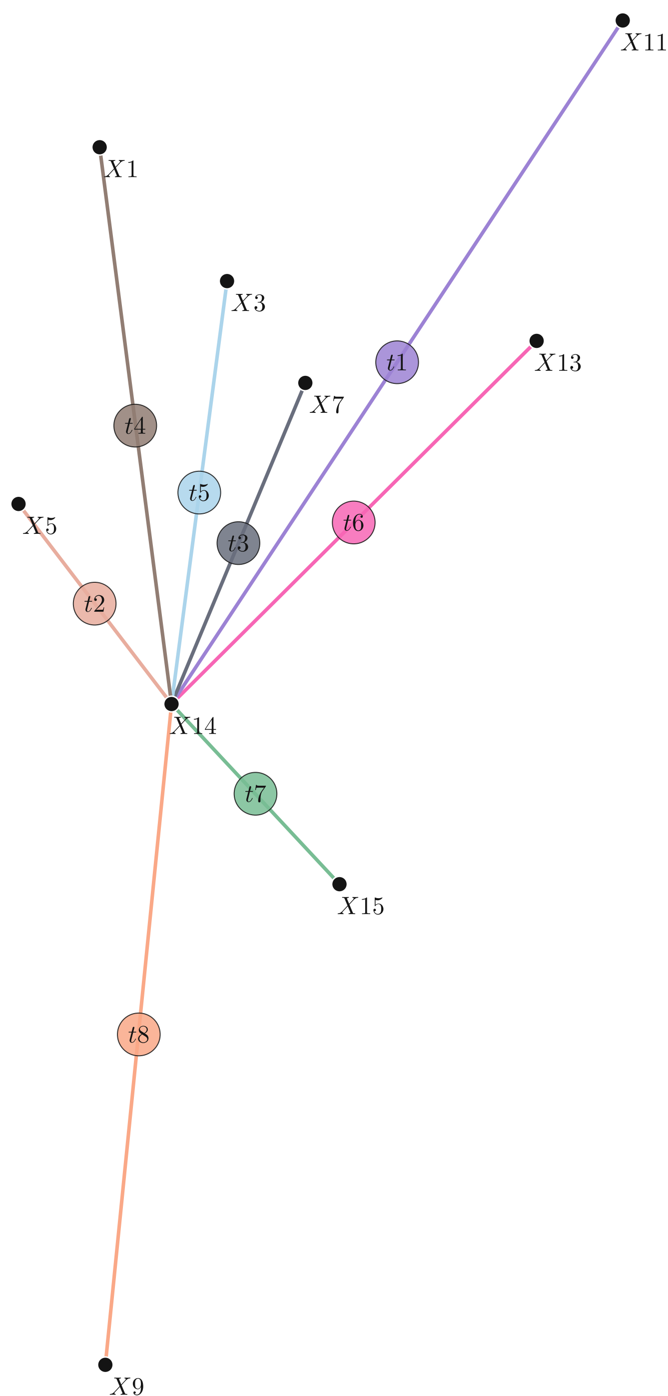

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



B.2.3 cycles.sql

```
SELECT t1.a, t2.b, t4.b, t5.b
FROM t1, t2, t3, t4, t5, t6, t7, t8
WHERE t1.a = t2.a
AND t2.a = t3.a
AND t3.a = t4.a
AND t4.a = t5.a
AND t5.a = t1.a
AND t6.a = t7.a
AND t7.a = t8.a
AND t8.a = t6.a
AND t7.a = t3.a;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

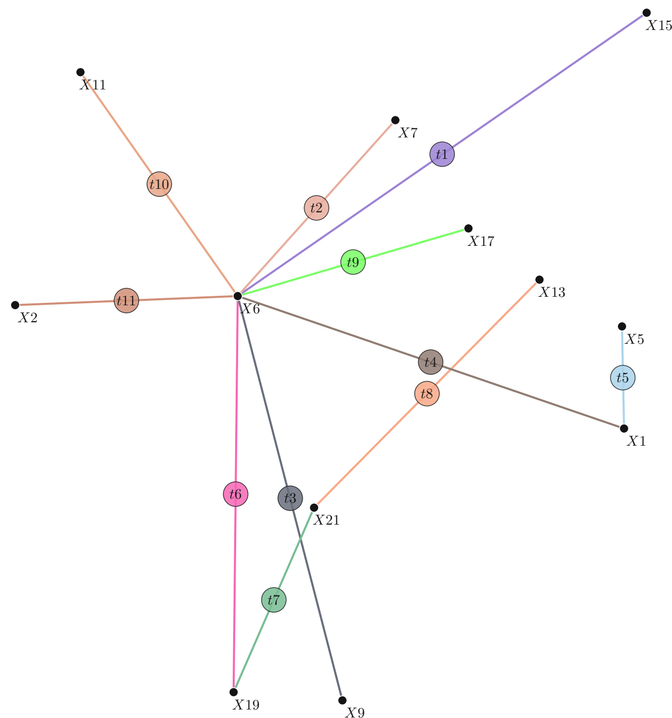


B.2.4 star.sql

```
select t1.a, t1.b, t3.b, t5.b, t6.b, t8.b
from t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

```
where t1.a = t2.a
and t2.a = t3.a
and t3.a = t4.a
AND t4.b = t5.a
AND t3.a = t6.a
AND t6.b = t7.a
AND t7.b = t8.a
AND t6.a = t9.a
AND t2.a = t10.a
AND t10.a = t11.b;
```



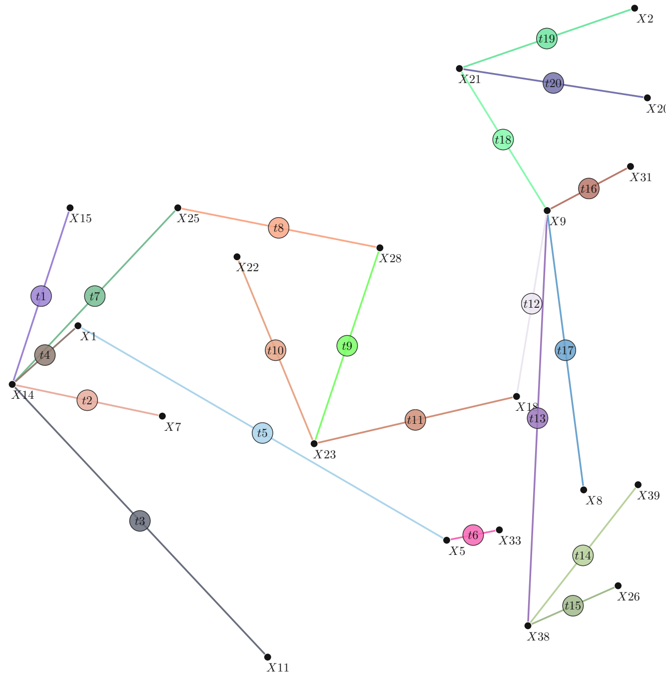
B.3 db3

B.3.1 tree.sql

```
select t1.a,t3.b,t15.b
from t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,
        t11, t12, t13, t14, t15, t16, t17, t18, t19, t20
where t1.a = t2.a
and t2.a = t3.a
and t3.a = t4.a
AND t4.b = t5.a
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
AND t5.b = t6.a
AND t7.a = t3.a
AND t8.b = t7.b
AND t9.a = t8.a
AND t10.b = t9.b
AND t11.a = t9.b
AND t12.a = t11.b
AND t13.a = t12.b
AND t14.a = t13.b
AND t15.b = t14.a
AND t16.a = t13.a
AND t17.b = t16.a
AND t18.b = t16.a
AND t19.b = t18.a
AND t20.b = t19.b;
```

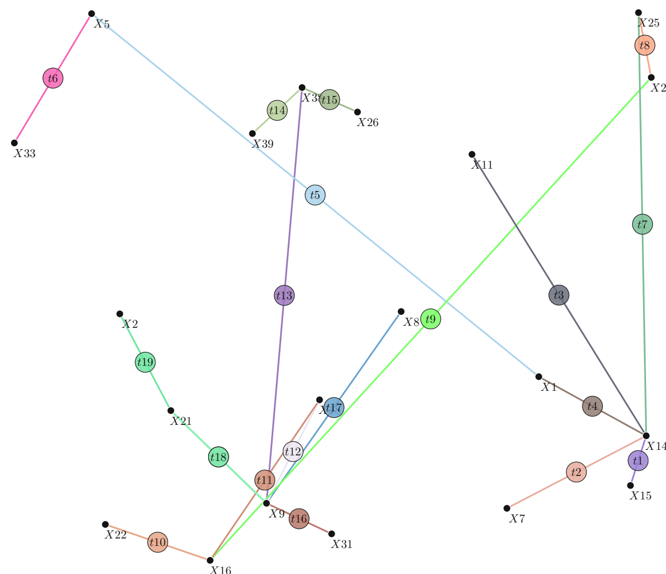


B.3.2 tree-2.sql

```
select t1.a,t3.b
from t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,
        t11, t12, t13, t14, t15, t16, t17, t18, t19, t20
where t1.a = t2.a
and t2.a = t3.a
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
and t3.a = t4.a
AND t4.b = t5.a
AND t5.b = t6.a
AND t7.a = t3.a
AND t8.b = t7.b
AND t9.a = t8.a
AND t10.b = t9.b
AND t11.a = t9.b
AND t12.a = t11.b
AND t13.a = t12.b
AND t14.a = t13.b
AND t15.b = t14.a
AND t16.a = t13.a
AND t17.b = t16.a
AND t18.b = t16.a
AND t19.b = t18.a
AND t20.b = t19.b
AND t20.a = t18.b;
```

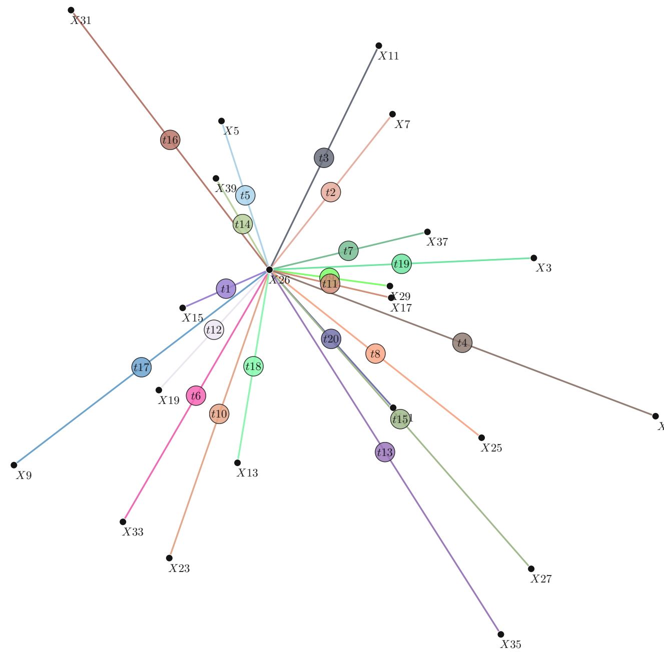


B.3.3 tree-3.sql

```
select t1.a,t3.b
from t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,
        11, t12, t13, t14, t15, t16, t17, t18, t19, t20
where t1.a = t2.a
and t2.a = t3.a
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
and t3.a = t4.a
AND t4.a = t5.a
AND t5.a = t6.a
AND t7.a = t3.a
AND t8.a = t7.a
AND t9.a = t8.a
AND t10.a = t9.a
AND t11.a = t9.a
AND t12.a = t11.a
AND t13.a = t12.a
AND t14.a = t13.a
AND t15.a = t14.a
AND t16.a = t13.a
AND t17.a = t16.a
AND t18.a = t16.a
AND t19.a = t18.a
AND t20.a = t19.a
AND t20.a = t18.a;
```

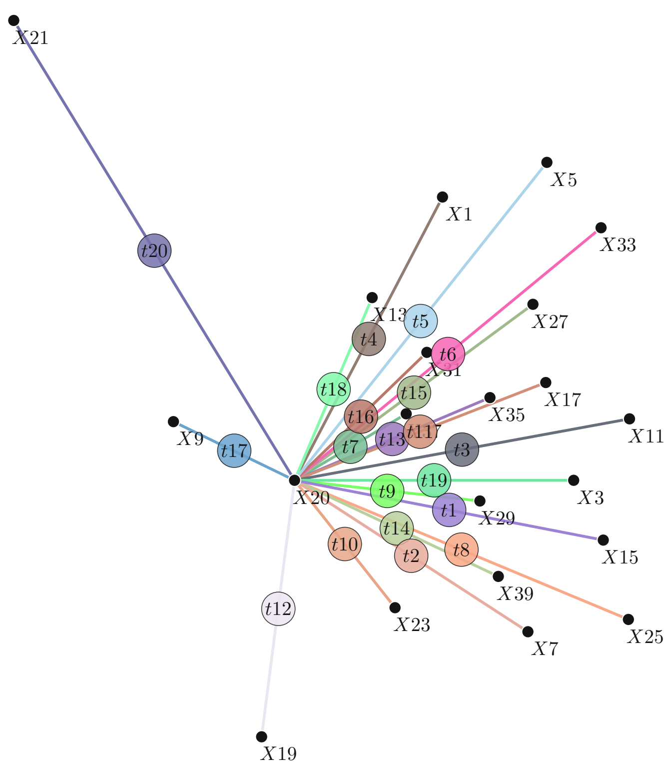


B.3.4 tree-4.sql

```
select t1.a,t3.b
from t1, t2, t3, t4, t5, t6, t7, t8, t9, t10,
        t11, t12, t13, t14, t15, t16, t17, t18, t19, t20
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
where t1.a = t2.a
and t2.a = t3.a
and t3.a = t4.a
AND t4.a = t5.a
AND t5.a = t6.a
AND t7.a = t3.a
AND t8.a = t7.a
AND t9.a = t8.a
AND t10.a = t9.a
AND t11.a = t9.a
AND t12.a = t11.a
AND t13.a = t12.a
AND t14.a = t13.a
AND t15.a = t14.a
AND t16.a = t13.a
AND t17.a = t16.a
AND t18.a = t16.a
AND t19.a = t18.a
AND t20.a = t19.a
AND t20.a = t18.a;
```

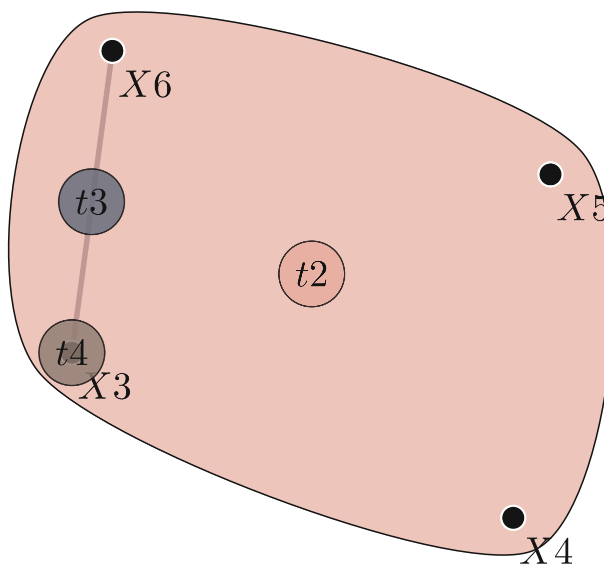


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

B.4 db4

B.4.1 query-1.sql

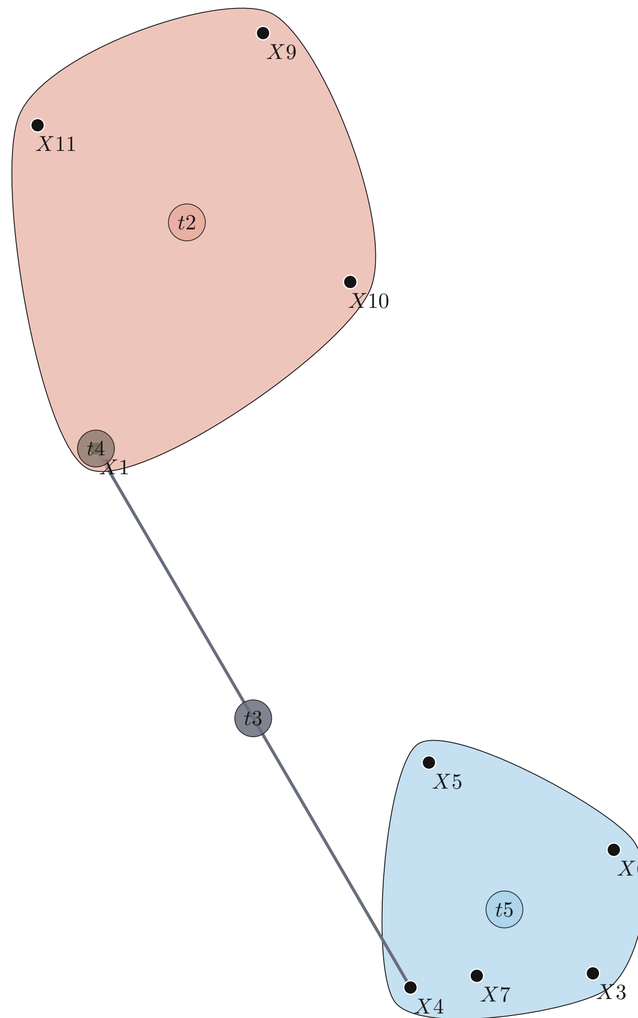
```
select * from t2, t3, t4
where t3.a = t4.a
and t2.a = t4.a
and t2.d = t3.b;
```



B.4.2 query-2.sql

```
select * from t2, t3, t4, t5
where t3.a = t4.a
and t2.a = t4.a
and t5.b = t3.b;
```

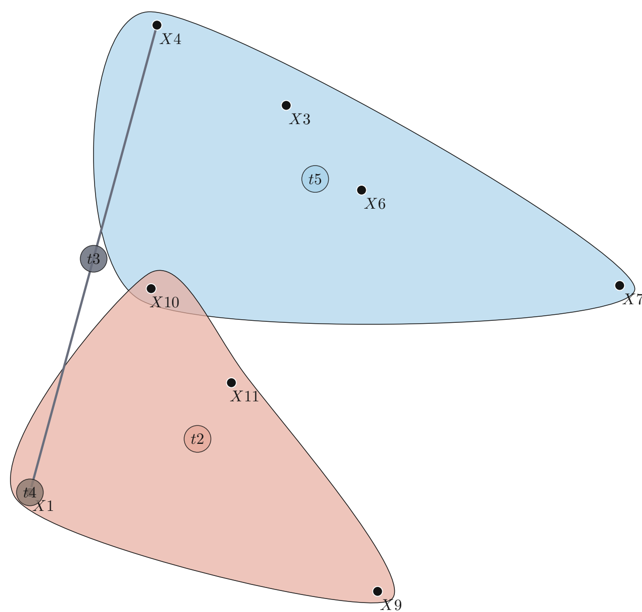

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

B.4.3 query-3.sql

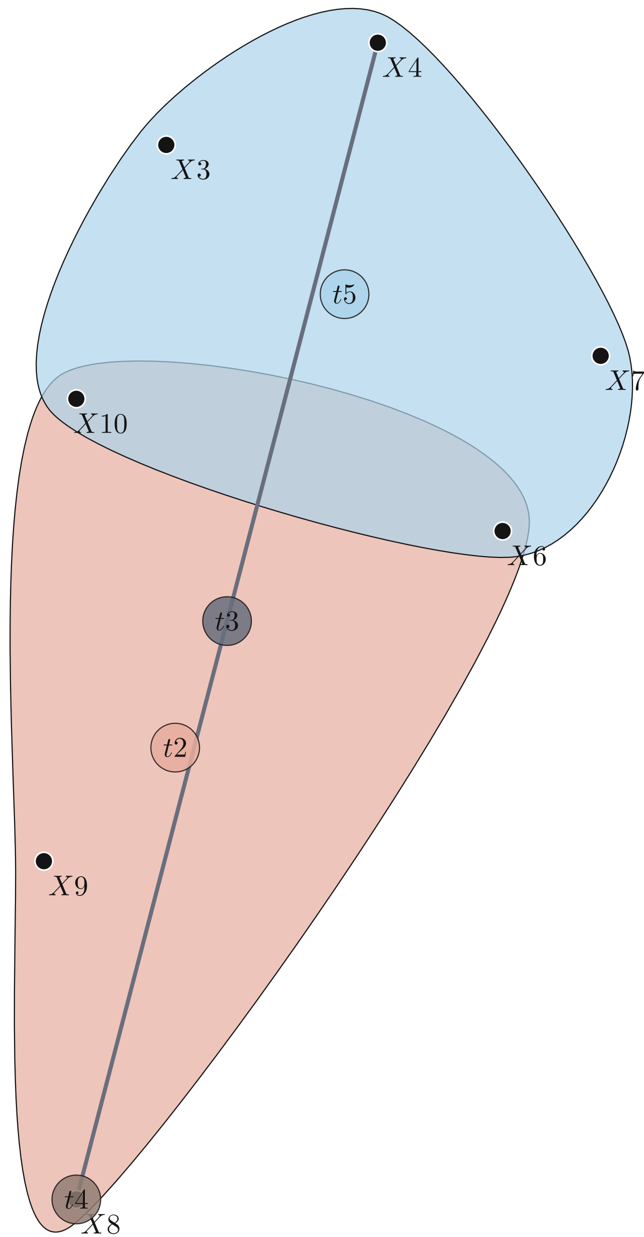
```
select * from t2, t3, t4, t5
where t3.a = t4.a
and t2.a = t4.a
and t5.b = t3.b
and t2.c = t5.c;
```



B.4.4 query-4.sql

```
select * from t2, t3, t4, t5
where t3.a = t4.a
and t2.a = t4.a
and t5.b = t3.b
and t2.c = t5.c
and t5.d = t2.d;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

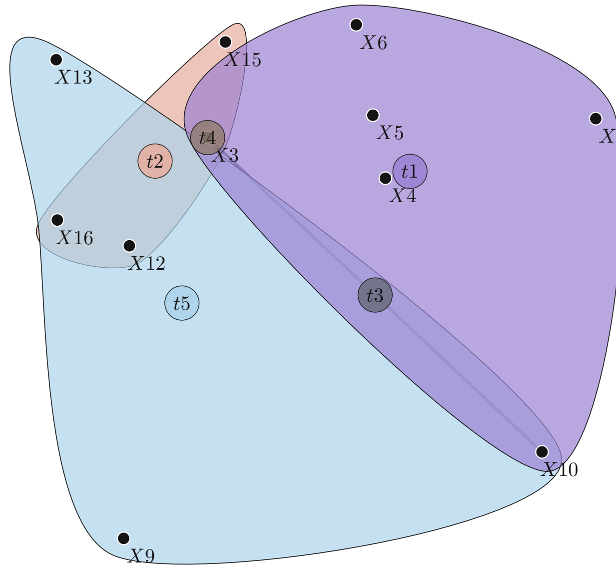


B.4.5 query-5-empty.sql

```
select * from t1, t2, t3, t4, t5
where t3.a = t4.a
and t2.a = t4.a
and t5.b = t3.b
and t2.c = t5.c
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
and t5.d = t2.d
and t5.b = t1.b
and t1.c = t4.a;
```

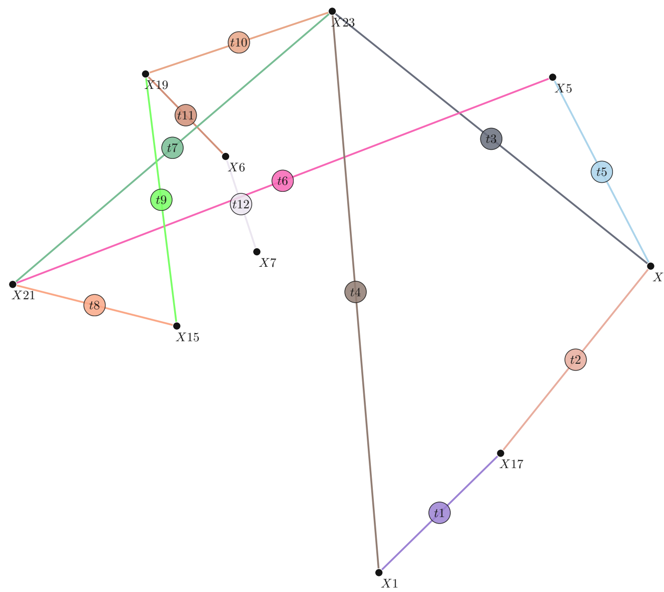


B.5 db5

B.5.1 query2.sql

```
select t1.a, t1.b, t2.b, t5.b, t3.b, t6.b, t8.b, t10.b, t11.b
from t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12
where t1.b = t2.a
and t2.b = t3.a
and t2.b = t5.a
and t3.b = t4.a
and t4.b = t1.a
and t5.b = t6.a
and t6.b = t7.a
and t6.b = t8.a
and t8.b = t9.a
and t9.b = t10.b
and t7.b = t3.b
and t10.a = t7.b
and t10.b = t11.a
and t12.a = t11.b;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



B.6 db6

B.6.1 query1.sql

```
select c1a.x, c1b.x, c3b.x, c5a.x
from c1a, c1b, c1c, c2a, c2b, c2c, c3a, c3b, c3c,
        c4a, c4b, c4c, c5a, c5b, c5c, c6a, c6b, c6c
where
c1a.x = c1b.y
and c1b.x = c1c.y
and c1c.x = c1a.y
and c2a.x = c2b.y
and c2b.x = c2c.y
and c2c.x = c2a.y
and c3a.x = c3b.y
and c3b.x = c3c.y
and c3c.x = c3a.y
and c4a.x = c4b.y
and c4b.x = c4c.y
and c4c.x = c4a.y
and c5a.x = c5b.y
and c5b.x = c5c.y
and c5c.x = c5a.y
and c6a.x = c6b.y
and c6b.x = c6c.y
and c6c.x = c6a.y
```

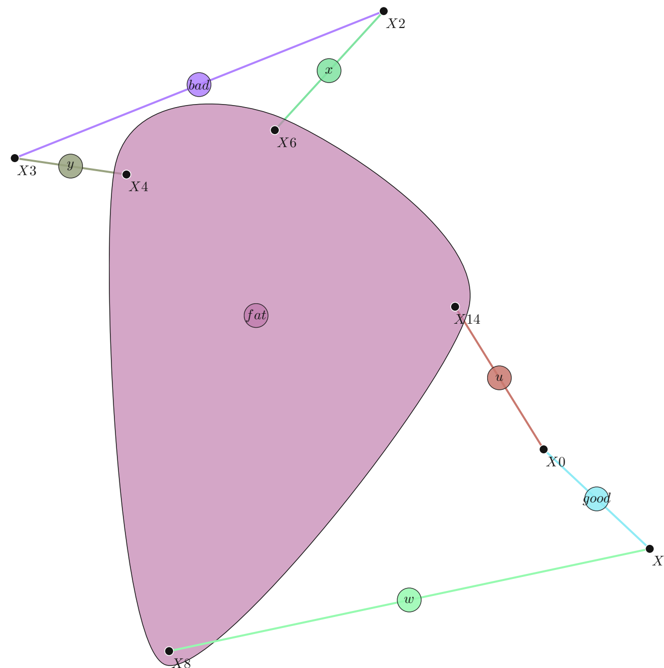

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

```
and c1a.z = c2b.z
and c2a.z = c3b.z
and c3a.z = c4b.z
and c4a.z = c5b.z
and c5a.z = c6b.z
and c5b.z = c6c.z;
```

B.7 fat_and_cycles

B.7.1 query-no-star.sql

```
select u.ugood, w.wgood, x.xbad, y.ybad, fat.d, fat.c, fat.b, fat.a from fa
natural join x
natural join y
natural join u
natural join w
natural join bad
natural join good;
```



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

B.8 tpc-h

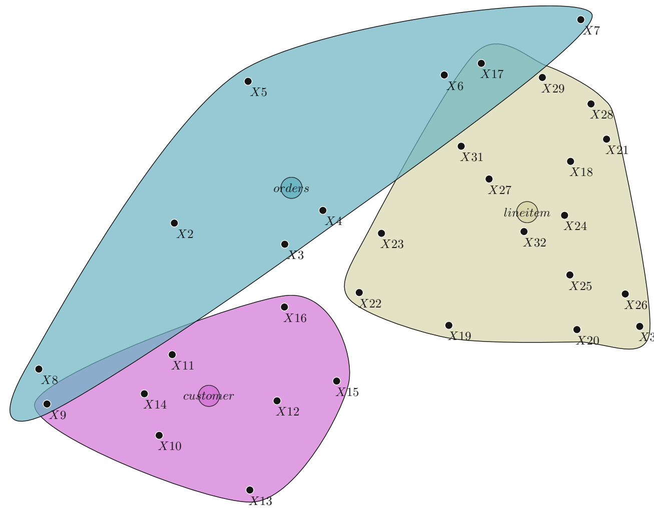
B.8.1 extra.sql

```
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount
from
customer,
orders,
lineitem,
supplier,
nation,
region,
part,
partsupp
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and p_partkey = l_partkey;
```

B.8.2 3-modified.sql

```
select
lineitem.l_orderkey,
lineitem.l_extendedprice,
lineitem.l_discount,
orders.o_orderdate,
orders.o_shippriority
from
customer,
orders,
lineitem
where c_custkey = o_custkey
and l_orderkey = o_orderkey;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



B.8.3 5-modified.sql

```
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount
from
customer,
orders,
lineitem,
supplier,
nation,
region
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey;
```

B.8.4 10-modified.sql

```
select
customer.c_custkey,
customer.c_name,
customer.c_acctbal,
nation.n_name,
customer.c_address,
customer.c_phone,
customer.c_comment
from
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
customer,
orders,
lineitem,
nation
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and c_nationkey = n_nationkey;
```

B.9 tpc-h-2

B.9.1 extra2.sql

```
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount
from
customer,
orders,
lineitem,
supplier,
nation,
region,
part
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and p_partkey = l_partkey;
```

B.9.2 extra3.sql

```
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount
from
customer,
orders,
lineitem,
supplier,
nation,
region,
part,
partsupp
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey
and p_partkey = l_partkey
and ps_partkey = p_partkey;
```

B.9.3 extra4.sql

```
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount, partsu
from
customer,
orders,
lineitem,
supplier,
nation,
region,
partsupp
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey;
```

B.9.4 extra2.sql

```
select nation.n_name, lineitem.l_extendedprice, lineitem.l_discount, partsu
from
customer,
orders,
lineitem,
supplier,
nation,
region,
partsupp
where c_custkey = o_custkey
and l_orderkey = o_orderkey
and l_suppkey = s_suppkey
and c_nationkey = s_nationkey
and s_nationkey = n_nationkey
and n_regionkey = r_regionkey;
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

Aberger, C. R., Lamb, A., Tu, S., Nötzli, A., Olukotun, K., and Ré, C. (2017). EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44.

Abseher, M., Musliu, N., and Woltran, S. (2017). Improving the efficiency of dynamic programming on tree decompositions via machine learning. *J. Artif. Intell. Res.*, 58:829–858.

Adler, I. (2004). Marshals, monotone marshals, and hypertree-width. *J. Graph Theory*, 47(4):275–296.

Adler, I., Gottlob, G., and Grohe, M. (2007). Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181.

Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T. L., and Washburn, G. (2015). Design and implementation of the LogicBlox system. In Sellis, T. K., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382. ACM.

Astrahan, M. M., Chamberlin, D. D., III, W. F. K., and Traiger, I. L. (1975). System R: A relational data base management system. In Hasselmeier, H. F. and Spruth, W. G., editors, *Data Base Systems, Proceedings, 5th Informatik Symposium, IBM Germany, Bad Homburg v. d. H., September 24-26, 1975*, volume 39 of *Lecture Notes in Computer Science*, pages 139–148. Springer.

Atserias, A., Grohe, M., and Marx, D. (2008). Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society.

Beeri, C., Fagin, R., Maier, D., Mendelzon, A. O., Ullman, J. D., and Yannakakis, M. (1981). Properties of acyclic database schemes. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*, pages 355–362. ACM.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chandra, A. K. and Merlin, P. M. (1977). Optimal implementation of conjunctive queries in relational data bases. In Hopcroft, J. E., Friedman, E. P., and Harrison, M. A., editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 77–90. ACM.

Chandra, A. K. and Stockmeyer, L. J. (1976). Alternation. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 98–108. IEEE Computer Society.

Chaudhuri, S. (1998). An overview of query optimization in relational systems. In Mendelzon, A. O. and Paredaens, J., editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B. J., Musliu, N., and Samer, M. (2008). Heuristic methods for hypertree decomposition. In Gelbukh, A. F. and Morales, E. F., editors, *MICAI 2008: Advances in Artificial Intelligence, 7th Mexican International Conference on Artificial Intelligence, Atizapán de Zaragoza, Mexico, October 27-31, 2008, Proceedings*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer.

Deshpande, A., Ives, Z. G., and Raman, V. (2007). *Adaptive Query Processing*, volume 1.

Donovan, A. A. and Kernighan, B. W. (2015). *The Go programming language*. Addison-Wesley Professional.

Dzulfikar, M. A., Fichte, J. K., and Hecher, M. (2019). The PACE 2019 parameterized algorithms and computational experiments challenge: The fourth iteration (invited paper). In Jansen, B. M. P. and Telle, J. A., editors, *14th International Symposium on Parameterized and Exact Computation, IPEC 2019, September 11-13, 2019, Munich, Germany*, volume 148 of *LIPIcs*, pages 25:1–25:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Fagin, R. (1983). Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550.

Fichte, J. K., Hecher, M., Lodha, N., and Szeider, S. (2018). An SMT approach to fractional hypertree width. In Hooker, J. N., editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 109–127. Springer.

Fischl, W. (2018). A guide to hyperbench. http://hyperbench.dbai.tuwien.ac.at/downloads/manual.pdf.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Fischl, W., Gottlob, G., Longo, D. M., and Pichler, R. (2019). Hyperbench: A benchmark and tool for hypergraphs and empirical findings. In Suciu, D., Skritek, S., and Koch, C., editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 464–480. ACM.

Fischl, W., Gottlob, G., and Pichler, R. (2018). General and fractional hypertree decompositions: Hard and easy cases. In den Bussche, J. V. and Arenas, M., editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 17–32. ACM.

Flum, J. and Grohe, M. (2006). Parameterized complexity theory (texts in theoretical computer science. an eatcs series).

Ghionna, L., Granata, L., Greco, G., and Scarcello, F. (2007). Hypertree decompositions for query optimization. In Chirkova, R., Dogac, A., Özsu, M. T., and Sellis, T. K., editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 36–45. IEEE Computer Society.

Ghionna, L., Greco, G., and Scarcello, F. (2011). H-DB: a hybrid quantitative-structural sql optimizer. In Macdonald, C., Ounis, I., and Ruthven, I., editors, *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 2573–2576. ACM.

Gottlob, G., Greco, G., Leone, N., and Scarcello, F. (2016). Hypertree decompositions: Questions and answers. In Milo, T. and Tan, W., editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 57–74. ACM.

Gottlob, G., Greco, G., and Scarcello, F. (2014). Treewidth and hypertree width. pages 3–38.

Gottlob, G., Lanzinger, M., Longo, D. M., Okulmus, C., and Pichler, R. (2020a). The HyperTrac project: Recent progress and future research directions on hypergraph decompositions. In Hebrard, E. and Musliu, N., editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21-24, 2020, Proceedings*, volume 12296 of *Lecture Notes in Computer Science*, pages 3–21. Springer.

Gottlob, G., Lanzinger, M., Pichler, R., and Razgon, I. (2020b). Complexity analysis of general and fractional hypertree decompositions. *CoRR*, abs/2002.05239.

Gottlob, G., Leone, N., and Scarcello, F. (1999). On tractable queries and constraints. In Bench-Capon, T. J. M., Soda, G., and Tjoa, A. M., editors, *Database and Expert Systems Applications, 10th International Conference, DEXA '99, Florence, Italy,*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

August 30 - September 3, 1999, Proceedings, volume 1677 of *Lecture Notes in Computer Science*, pages 1–15. Springer.

Gottlob, G., Leone, N., and Scarcello, F. (2001a). The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498.

Gottlob, G., Leone, N., and Scarcello, F. (2001b). Hypertree decompositions: A survey. In Sgall, J., Pultr, A., and Kolman, P., editors, *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Marianske Lazne, Czech Republic, August 27-31, 2001, Proceedings*, volume 2136 of *Lecture Notes in Computer Science*, pages 37–57. Springer.

Gottlob, G., Leone, N., and Scarcello, F. (2002). Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627.

Gottlob, G., Okulmus, C., and Pichler, R. (2020c). Fast and parallel decomposition of constraint satisfaction problems. In Bessiere, C., editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1155–1162. ijcai.org.

Gottlob, G. and Samer, M. (2008). A backtracking-based algorithm for hypertree decomposition. *ACM J. Exp. Algorithmics*, 13.

Graham, M. (1980). *On the universal relation.*

Grohe, M. and Marx, D. (2006). Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 289–298. ACM Press.

Harvey, P. and Ghose, A. (2003). Reducing redundancy in the hypertree decomposition scheme. In *15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2003), 3-5 November 2003, Sacramento, California, USA*, pages 474–481. IEEE Computer Society.

Jain, S., Moritz, D., Halperin, D., Howe, B., and Lazowska, E. (2016). SQLShare: Results from a multi-year SQL-as-a-service experiment. In Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 281–293. ACM.

Khayyat, Z., Lucia, W., Singh, M., Ouzzani, M., Papotti, P., Quiané-Ruiz, J., Tang, N., and Kalnis, P. (2015). Lightning fast and space efficient inequality joins. *Proc. VLDB Endow.*, 8(13):2074–2085.

Leis, V., Gubichev, A., Mirchev, A., Boncz, P. A., Kemper, A., and Neumann, T. (2015). How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Michael Abseher, Frederico Dusberger, N. M. and Woltran, S. (2017). Tree decomposition features.

Musliu, N. (2008). An iterative heuristic algorithm for tree decomposition. In Cotta, C. and van Hemert, J. I., editors, *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, volume 153 of *Studies in Computational Intelligence*, pages 133–150. Springer.

Ngo, H. Q., Ré, C., and Rudra, A. (2013). Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16.

Paquette, J. and Tokuyasu, T. A. (2011). Hypergraph visualization and enrichment statistics: how the EGAN paradigm facilitates organic discovery from big data. In Rogowitz, B. E. and Pappas, T. N., editors, *Human Vision and Electronic Imaging XVI, part of the IS&T-SPIE Electronic Imaging Symposium, San Francisco Airport, California, USA, January 24-27, 2011, Proceedings*, volume 7865 of *SPIE Proceedings*, page 78650E. SPIE/IS&T.

Postgres (2020). Postgres 13 documentation. `https://www.postgresql.org/docs/13/`.

Robertson, N. and Seymour, P. D. (1986). Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322.

Russell, S. and Norvig, P. (2002). Artificial intelligence: a modern approach.

Scarcello, F., Greco, G., and Leone, N. (2007). Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci.*, 73(3):475–506.

Schidler, A. and Szeider, S. (2020). Computing optimal hypertree decompositions. In Blelloch, G. E. and Finocchi, I., editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020, Salt Lake City, UT, USA, January 5-6, 2020*, pages 1–11. SIAM.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. (1979). Access path selection in a relational database management system. In Bernstein, P. A., editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34. ACM.

Stonebraker, M. and Rowe, L. A. (1986). The design of postgres. In Zaniolo, C., editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, pages 340–355. ACM Press.

Stonebraker, M., Rowe, L. A., and Hirohama, M. (1990). The implementation of postgres. *IEEE Trans. Knowl. Data Eng.*, 2(1):125–142.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Tarjan, R. E. and Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579.

Transaction Processing Performance Council (2014). TPC-H, a decision support benchmark. http://www.tpc.org/tpch/default.asp.

Tu, S. and Ré, C. (2015). Duncecap: Query plans using generalized hypertree decompositions. In Sellis, T. K., Davidson, S. B., and Ives, Z. G., editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2077–2078. ACM.

Utesch, M. (1999). *Ein Beitrag zur Anfrageoptimierung in Datenbanksystemen mit genetischen Algorithmen*. VDE-Verlag.

Valdivia, P., Buono, P., Plaisant, C., Dufournaud, N., and Fekete, J. (2021). Analyzing dynamic hypergraphs with parallel aggregated ordered hypergraph visualization. *IEEE Trans. Vis. Comput. Graph.*, 27(1):1–13.

Valiente, G. (2013). *Algorithms on trees and graphs*. Springer Science & Business Media.

Vardi, M. Y. (1982). The complexity of relational query languages (extended abstract). In Lewis, H. R., Simons, B. B., Burkhard, W. A., and Landweber, L. H., editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM.

Yannakakis, M. (1981). Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society.

Yu, C. T. and Özsoyoglu, M. Z. (1979). An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979, 6-8 November, 1979, Chicago, Illinois, USA*, pages 306–312. IEEE.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.