TU WIEN Informatics

Improving Joins in Pandas

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Alexander Steindl, BSc
Matrikelnummer 01529262

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Dr.techn. Matthias Lanzinger

Wien, 20. April 2023

Alexander Steindl	Reinhard Pichler

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

TU WIEN Informatics

Improving Joins in Pandas

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Alexander Steindl, BSc
Registration Number 01529262

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Dr.techn. Matthias Lanzinger

Vienna, 20th April, 2023

Alexander Steindl	Reinhard Pichler

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

Erklärung zur Verfassung der Arbeit

Alexander Steindl, 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, 20. April 2023

Alexander Steindl

Acknowledgements

First of all, I would like to thank my supervisor Reinhard Pichler and my co-supervisor Matthias Lanzinger for the helpful feedback I received during the work on this thesis. This has enabled me to improve my work considerably. Beyond that, I am especially grateful for the responsiveness throughout our email discussions and the flexibility in scheduling meetings.

Finally, I would like to thank both my brother and my parents for their continued support throughout my studies.

Kurzfassung

Pandas ist ein unter Datenwissenschaftlern sehr beliebtes Werkzeug zur Datenanalyse und -manipulation. Datenwissenschaftler müssen häufig Rohdaten aus potenziell heterogenen Datenquellen kombinieren. Derzeit verbringen sie bis zu 90 % ihrer Zeit mit dem Bereinigen, Integrieren und Transformieren von Daten. Pandas bietet zwar bestimmte Funktionalitäten eines Datenbanksystems, z.B. unterstützt es mit seiner DataFrame-Abstraktion eine tabellarische Datendarstellung und einige gängige Operationen darauf, aber die wichtigste Datenbankoperation – die Ausführung von Joins – wird derzeit nur sehr rudimentär unterstützt. Es werden nur zweiseitige Joins unterstützt, wobei der Benutzer die Join-Reihenfolge angeben muss, wenn mehr als zwei Relationen kombiniert werden sollen. Je nach Join-Reihenfolge kann die Ausführungszeit jedoch drastisch variieren.

Herkömmliche Datenbanksysteme verwenden Abfrageoptimierer, um eine bessere Verknüpfungsreihenfolge zu finden. Diese Optimierer berücksichtigen jedoch nicht die strukturellen Eigenschaften einer Abfrage und verlassen sich stattdessen auf Kardinalitätsschätzungen der an einer Abfrage beteiligten Relationen. Im Laufe der Jahre haben Forscher der Datenbanktheorie dekompositionsbasierte Methoden für die Abfrageauswertung vorgeschlagen. Während die Auswertung konjunktiver Abfragen im Allgemeinen NP-vollständig ist, wurde gezeigt, dass es Algorithmen gibt, die die Unterklasse der azyklischen Abfragen und eine Verallgemeinerung davon in polynomieller Zeit beantworten können. Darüber hinaus wurden für das Problem der exponentiellen Zwischenergebnisgrößen bei Join-Abfragen sogenannte "worst-case optimal join" Algorithmen vorgeschlagen. Sie können dadurch kategorisiert werden, dass ihre Zeitkomplexität immer innerhalb der größtmöglichen Ausgabengröße liegt.

Wir erweitern Pandas um eine Funktion, die eine Liste von DataFrame-Objekten akzeptiert und einen natürlichen Join durch Anwendung verschiedener Optimierungen aus der Datenbanktheorie durchführt, abhängig von der Art der Join-Abfrage. Wir konzentrieren uns auf azyklische Abfragen und die "Dreiecks"-Abfrage als einfachste Form einer zyklischen Abfrage, die beide in der Praxis sehr häufig vorkommen, wobei letztere hauptsächlich in Graphdatenbanken zu finden ist. Unsere empirische Evaluierung hat gezeigt, dass wir die meisten azyklischen Abfragen mit dem sogenannten Yannakakis-Algorithmus deutlich besser ausführen können als mit einer Basislösung. Allerdings

konnten wir die Ausführungszeit nur für eine von drei Dreiecksabfragen mit einem auf Partitionierung basierenden "worst-case optimal join" Algorithmus verbessern.

Abstract

Pandas is a very popular tool among data scientists for data analysis and manipulation. Data scientists often need to combine raw data from potentially heterogeneous data sources. Currently, they spend up to 90 % of their time cleaning, integrating and transforming data. While Pandas provides certain functionalities of a database system, e.g. it supports a tabular data representation with its DataFrame abstraction and some common operations on it, the most important database operation, namely the execution of joins, is currently only supported in a very rudimentary way. Only two-way joins are supported, with the user having to specify the join order if more than two relations are to be combined. However, depending on the join order, the execution time can diverge drastically.

Traditional database systems use query optimizers to find a better join order. However, these optimizers do not take the structural properties of a query into account, instead relying on cardinality estimates of the relations involved in a query. Over the years, database theory researchers have proposed decomposition-based methods for query evaluation. While conjunctive query evaluation is in general NP-complete, it has been shown that there are algorithms that can answer the subclass of acyclic queries and a generalization thereof in polynomial time. Furthermore, worst-case optimal join algorithms have been proposed for the problem of exponential intermediate result sizes in join queries. They can be categorized by the fact that their time complexity is always within the largest possible output size.

We extend Pandas with a function that accepts a list of DataFrame objects and performs a natural join by applying different optimizations from database theory, depending on the type of join query. We focus on acyclic queries and the "triangle" query as the simplest form of a cyclic query, both of which are very common in practice, the latter being found mainly in graph databases. Our empirical evaluation shows that we can execute most acyclic queries significantly better with the so-called Yannakakis algorithm, compared to a baseline. However, we were only able to improve the execution time for one out of three triangle queries using a partitioning-based worst-case optimal join algorithm.

Contents

CHAPTER 1

Introduction

One of the most important trends in database research in recent years has been to address challenges faced within the data science field. Data science is about creating pipelines that leverage raw data from potentially heterogeneous data sources with varying data quality to gain new insights by using data analytics techniques. Most of the time is currently spent on data cleansing, integration, and transformation, which together account for 80-90%. These are problems that the database community has been dealing with for decades in the context of enterprise data. Consequently, the database community has much to offer ML users with its expertise in these areas [AAA+22].

1.1 Problem Statement and Motivation

Data scientists use a rich ecosystem of open source libraries to get their work done. Pandas[1] is a very popular tool that allows both data analysis and manipulation. Due to its flexibility and performance, it is used in a variety of academic and commercial fields, including finance, statistics and others. Pandas also provides certain functionalities of a database management system (DBMS). In fact, it supports the tabular representation of data with its DataFrame abstraction, which provides some operations known from a DBMS, such as filtering and aggregation.

However, the most important database operation, namely the execution of joins, which is important for the integration of data, is only supported in a very rudimentary way. This is because currently only two-way joins are supported, and the user has to manually specify the join order if more than two relations are to be joined, which is also a common use case in data science. As a result, the user has to guess, so to speak, which order is better from a performance point of view, and this is a difficult problem. Traditional relational DBMS solve this problem by relying on query optimizers that apply a set of

[1] https://pandas.pydata.org

optimizations that are primarily based on cardinality estimates of the relations involved in a query. In this way, efficient query execution plans can be generated for a wide variety of queries. A query execution plan is a tree-structured representation of the relations and concrete operators needed to evaluate a given query. It also defines the order in which the individual joins are executed. Thus, a serious shortcoming of Pandas is the lack of such optimizations for the query evaluation process.

1.2 Aim of the Thesis and Expected Results

Motivated by this problem, we would like to improve the current situation in the framework of this thesis. In this thesis, we will concentrate on two specific forms of join queries: acyclic queries and so-called "triangle queries" as the most basic form of cyclic queries. Fischl et al. [FGLP21] have analysed query logs and benchmarks with thousands of relational queries, and they have found out that most join queries are acyclic. Similarly, Bonifati et al. [BMT20] have analysed millions of queries from query logs of public SPARQL endpoints, and they have confirmed that most queries are indeed acyclic. Beyond that, Aberger et al. [ALOR17, ALT+17] state that, especially in graph databases, the triangle query - despite its simplicity - constitutes an important case of cyclic queries.

For both types of queries, there exist structure-based approaches in database theory to evaluate them, i.e. decomposition-based approaches for acyclic queries and worst-case optimal join algorithms for the triangle query. Both Pandas and traditional DBMS ignore the possibilities these approaches offer at this point.

The aim of this thesis is to investigate whether these algorithms that have been shown to exhibit better runtime behaviour in theory can also be deployed effectively in practice and, in particular, in the context of Pandas. This will involve extending Pandas with a function that allows the user to specify a list of DataFrame objects to be joined with a *natural join*, i.e. the join is performed based on common column names. A natural join as such does not represent a restriction of generality compared to an equi-join, which uses only equality conditions combined with logical conjunction, since it is always possible to convert an equi-join into a natural join by renaming the attributes and possibly the relations. Depending on the type of the join query, a different algorithm known from database research is applied with the intent to improve the execution time. In case of a cyclic join involving more than three relations – i.e., in particular, not being a "triangle query" – the join is conducted without optimization as a sequence of two-way joins.

Our enhancement is intended to make Pandas an even stronger tool in the field of data science. Moreover, we want to empirically validate the benefits of integrating the chosen algorithms from database theory into Pandas.

1.3 Structure of the Thesis

The thesis is structured as follows. First, we provide some basic definitions in Chapter 2. Then, we discuss structural-decomposition based approaches for acyclic queries and a generalization thereof in Chapter 3. Subsequently, we define worst-case optimality for join algorithms based on the so-called AGM bound and present two algorithms fulfilling the criterion in Chapter 4. Next, we look at the Python ecosystem of tools for Data Science with a focus on Pandas as the currently most popular library for data preparation in Chapter 5. Then, we discuss related work and outline what distinguishes this work in Chapter 6. After that, we describe the system architecture of our software prototype in Chapter 7, followed by an empirical evaluation in Chapter 8. Finally, we conclude the thesis with Chapter 9.

CHAPTER 2

Basic definitions

In this chapter we will lay the foundation for the topics covered in this thesis by providing some basic definitions. First, we will introduce hypergraphs, which are a generalization of graphs that enable the representation of queries. Then we will define conjunctive queries, one of the most fundamental forms of queries in a database system. Next, we will briefly discuss traditional query optimization techniques used by relational database management systems, which aim to find the most efficient execution plan for a given query. Finally, we will give a short overview of join algorithms supported by common database systems.

2.1 Hypergraphs

A hypergraph $\mathcal{H} = (V, E)$ is composed of a set of vertices V and a set of edges E. In a hypergraph, an edge $e \in E$ consists of a non-empty subset of V, i.e. any number of nodes may be part of it. This property distinguishes it from a graph in which exactly two vertices are part of an edge. To emphasize this difference, the edges of a hypergraph are also referred to as hyperedges in the literature [GGLS16].

Acyclic hypergraphs are useful for many real-world scenarios, such as query evaluation, while being relatively simple and having many desirable properties that make them easier to work with than other hypergraphs. There are several notions [Fag83] for hypergraph acyclicity. In this thesis, we will use α-acyclicity as it is the most general one. Before we can define it formally, we need to introduce a few more relevant terms. The so-called primal graph [GGLS16] $G(\mathcal{H}) = (V, E')$ of a hypergraph $\mathcal{H}$ can be constructed by reusing the same vertices V, but using as edges all vertex pairs $\{u, v\}$ that are contained in at least one hyperedge. A sequence of distinct vertices $v_0, v_1, \ldots, v_k$ is termed a path, if it holds that v_i and v_{i+1} are adjacent for all $0 \leq i < k$, i.e. $\{v_i, v_{i+1}\} \in E'$. A cycle is defined as path $v_0, v_1, \ldots, v_k$ for some $k \geq 2$, where the start and end vertices of the path are adjacent, i.e. $\{v_0, v_k\} \in E'$. A clique $C \subseteq V$ contains pairwise adjacent vertices, i.e.

for every $i \neq j$ it holds that $\{v_i, v_j\} \in E'$, if $\{v_i, v_j\} \subset C$. A hypergraph $\mathcal{H}$ is considered *α-acyclic* if it satisfies two conditions. First, the hypergraph $\mathcal{H}$ has to be *conformal*, i.e. every clique C of G is contained in a hyperedge of $\mathcal{H}$. Second, the graph G has to be *chordal*. This condition is satisfied, if there is at least one "chord" edge that connects two non-consecutive vertices in a cycle for every cycle with four or more vertices [TY84].

2.2 Conjunctive queries

Conjunctive queries only use the logical conjunction operator, i.e. disjunctions and negations are excluded. According to Gottlob et al. [GGLS16], such a query Q can be represented as a Datalog rule with the following structure, where $r_1, \ldots, r_n$ are relation symbols and $u, u_1, \ldots, u_n$ are lists of terms, i.e. they can be variables or constants.

$$out(u) \leftarrow r_1(u_1), \ldots, r_n(u_n).$$

The left-hand side of the rule specifies the list of terms that shall be part of the output of the query. In case u is empty, Q is referred to as a *boolean* conjunctive query. The right-hand side of the rule defines which atoms are part of the query, i.e. $atoms(Q) = \{r_1(u_1), \ldots, r_n(u_n)\}$ [GGLS16].

The expressive power of conjunctive queries is equivalent to non-nested SQL queries with a `SELECT-FROM-WHERE` structure and the restriction that the `WHERE` clause only consists of conjunctions and equality constraints. Chandra and Merlin [CM77] have proven that the evaluation of a *boolean* conjunctive query over some database instance is NP-complete in general. However, as we will see later on, there are subclasses of queries that can be evaluated efficiently [GLS01a].

Given a query Q, we can construct a corresponding hypergraph $\mathcal{H} = (V, E)$ in a very intuitive way. The vertices V correspond to the variables occurring in the atoms of the query, i.e. the variables from the lists $u_1, \ldots, u_n$. For every atom of the query that contains at least one variable, we then add a hyperedge to the set E that contains all variables of this very atom. There are no other hyperedges present in E [GGLS16].

Later we will also refer to join queries. These are a subset of conjunctive queries where only the join operation is used, while projection and selection operations are excluded.

2.3 Traditional query optimization

A relational DBMS relies on the two key components *query execution engine* and *query optimizer* for the evaluation of SQL queries [Cha98].

A *query execution engine* supports a number of physical operators. These include relation access operators, such as sequential scan and index scan, and join operators, that will be discussed in Section 2.4, among others. Each operator takes as input one or more data

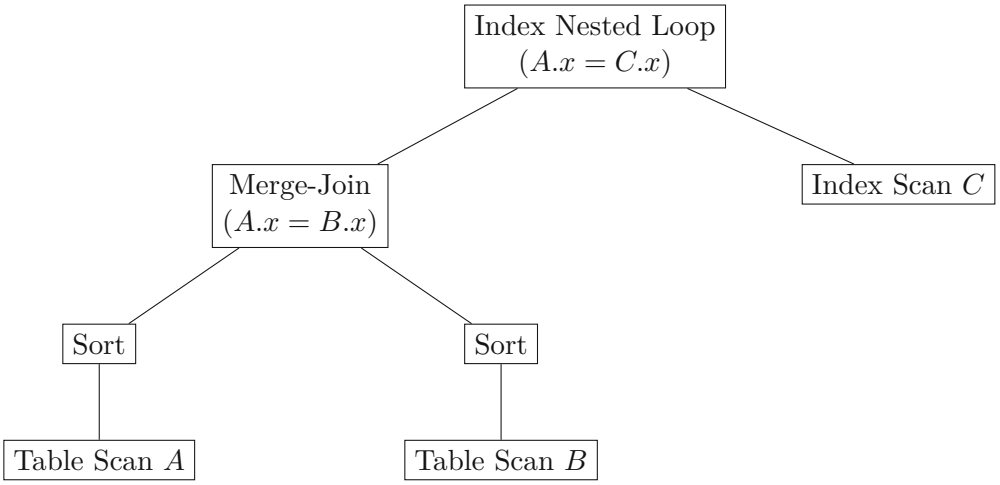


Figure 2.1: Query execution plan (based on [Cha98])

streams and produces an output data stream. Intuitively, physical operators represent chunks of code that serve as basic building blocks for query execution. Each query execution can be visualized by a physical operator tree, which is later also referred to as a query execution plan. The tree consists of the physical operators as vertices and the edges indicate the data flow between the operators. An example of a query execution plan is shown in Figure 2.1. Given a query execution plan, the query execution engine executes the plan and thereby produces a result set for the query [Cha98].

Given a parsed representation of a SQL query, the goal of a *query optimizer* is to generate an efficient execution plan for it, which can then be passed to the query execution engine for execution. The task of finding an efficient execution plan is non-trivial, since the search space can include a vast number of possible execution plans, and the response time for executing the plans can vary significantly. The search space depends on (1) the physical operators supported by the target query execution engine, and (2) the set of algebraic transformations we can perform for a relational algebraic representation of a query while preserving the equality of them. The latter factor involves changing the order of the joins by exploiting the commutative and associative nature of the join operation. To find an efficient query execution plan, we must use a cost model that assigns an estimated cost to each partial or complete plan in the search space. Each operator of a plan is assigned an estimated cost for the required CPU resources, I/O and/or communication costs incurred during its execution. To this end, the database system typically maintains statistics on relations and indices, such as the number of unique values in a column, the number of tuples in a relation, and/or histograms. The search process is performed in a bottom-up method using a dynamic programming approach. First, it finds the best plans for 1-table scans, i.e. the physical operators are assigned for accessing the relations involved in the query. Then, it constructs 2-table joins, 3-table joins, and so on, until it find the best plans for complete query execution plans [Cha98].

A major challenge for a query optimizer is to find a good join order, because it depends on estimating the output cardinality of each join operation. These estimates serve as predicted sizes for subsequent operator inputs until the complete query execution plan is estimated in terms of cost. However, inaccurate estimates, partly caused by attribute correlation, can often lead to a significant cost error. Sometimes the query optimizer has to choose between a "conservative" plan, which works quite well in many scenarios, and a more "aggressive" plan, which works better when estimates are accurate, but can result in much worse performance when the estimate is slightly off [DIR07].

2.4 Join algorithms

According to Elmasri et al. [EN15], we distinguish four different physical operators for performing a join. We discuss them in the context of the join $R \bowtie_{R.A=S.B} S$.

- **Nested-loop join**: This is the standard algorithm that does not require any indices. The outer loop retrieves a tuple $r \in R$, and the inner-loop then searches for a matching tuple $s \in S$ such that $r[A] = s[B]$.
- **Index-based nested-loop join**: If there is an index for any of the two join attributes, we can iterate over one relation as before, but could then use the index of the join attribute of the other relation to search directly for matching tuples.
- **Sort-merge join**: If the tuples in R and S are physically sorted by the values of their corresponding join attribute, we can perform the join in the most efficient way. We can then scan both relations simultaneously and find matching tuples. If the relations are not yet sorted, we could perform an external sorting prior to performing the join. If the values of at least one of the two join attributes are unique, only one scan per relation is required.
- **Hash-join**: In this case, the tuples of the relations R and S are partitioned into smaller buckets according to their join attribute. To determine the bucket for a tuple, a hashing function h is applied to the join attribute A of relation R and B for relation S. As part of the *partitioning phase*, we would scan over the smaller relation (e.g., R) and partition it such that all tuples with the same hash value (i.e. $h(A)$) are assigned to the same hash bucket. We assume that this relation can be kept entirely in memory. During the *probing phase*, we would then scan the other relation (i.e. S), compute the hash value (i.e. $h(B)$) for the tuple, and then *probe* the tuples assigned to the corresponding bucket of R and combine the tuple from S with all matching tuples from R.

CHAPTER 3

Decomposition-based query evaluation

In this chapter we present structural-decomposition based approaches for query evaluation. These do not take into account the cardinality estimates as a query optimizer of a traditional DBMS does. Instead, we utilize the structural properties of the query. We start this chapter by defining acyclic queries and explaining how they can be recognized. Next, we introduce the Yannakakis algorithm, which can efficiently evaluate acyclic queries. Then, we will go beyond acyclic queries and discuss how we can generalize this class of queries to a broader class of nearly acyclic queries. Finally, we will present one possible way for evaluating such queries.

3.1 Acyclic queries

A join query involving multiple relations can potentially result in an exponentially large output. Furthermore, the problem of checking whether the output of such a query is empty is NP-complete. In general, efficient algorithms for joining may not exist, unless P = NP [Yan81]. However, this does not apply to acyclic queries. In case of acyclic queries, the associated hypergraph is itself acyclic. The definition of an acyclic hypergraph given in Chapter 2 is equivalent to the existence of a join tree [GLS01a].

A join tree [BFMY83] is a tree $T = (V, E)$, where V consists of the atoms of a query, i.e. $V = \{R_1(\vec{z_1}), R_2(\vec{z_2}), \ldots, R_n(\vec{z_n})\}$. Beyond that, the tree has to satisfy the following two conditions.

1. Every edge $\{R_i, R_j\} \subseteq E$, where $i \neq j$, is labelled with its common variables, $z_i \cap z_j$.
2. For every vertex pair (R_i, R_j) with $i \neq j$, every shared variable $z \in z_i \cap z_j$ is part of the label of every edge on the unique path between R_i and R_j. Thus, z induces

a connected subtree within T. As a result, this property is also referred to as the *connectedness condition* in the literature (e.g. [GLS01b]).

The join tree can be used to evaluate acyclic queries, previously referred to as tree queries by Bernstein and Chiu [BC81], in polynomial time [Yan81]. This is because acyclic queries can rely on semi-joins to compare subtuples of common attributes between two relations, resulting in less data transfer if the relations reside at different sites and improved query performance. Conversely, cyclic queries require more data transfer [YO79].

Bernstein and Chiu [BC81] proposed an algorithm to determine whether a query is a tree query. Yu et al. [YO79] later identified two limitations with the algorithm. First, it is only applicable to a subset of acyclic queries, due to the assumption that semi-joins would only be able to compare one common attribute between two relations. Second, Bernstein and Chiu's algorithm may produce false negatives in some cases, i.e., it may incorrectly classify a query as cyclic.

To overcome these limitations, Yu et al. have created an algorithm that operates in linear time and requires linear space, which is referred to in the literature as GYO (Graham, Yu, and Ozsoyoglu) [Gra80, YO79] reduction. The algorithm eliminates vertices and edges in the hypergraph of the query, while preserving the query type. If the resulting hypergraph is a "null query graph", with no edges or vertices, the query is acyclic; otherwise, it is cyclic. In case the query is acyclic, we can use the elimination sequence to construct a join tree [YO79].

The GYO reduction works as follows for a query Q of the form:

$$out() : -R_1(\vec{z_1}), R_2(\vec{z_2}), \ldots, R_n(\vec{z_n}).$$

The hypergraph corresponding to Q is defined as $\mathcal{H} = (V, E)$, where V is the set of variables and $E = \{\vec{z_1}, \vec{z_2}, \ldots, \vec{z_n}\}$ is the set of edges.

The algorithm repeatedly applies the following two rules until no more eliminations can be done.

1. **Elimination of isolated hyperedges**: A hyperedge $\vec{z_i}$ is eliminated from E, if each vertex in $\vec{z_i}$ does not appear in any other hyperedge.
2. **Elimination of redundant hyperedges**: A hyperedge $\vec{z_i}$ is eliminated from E, if there exists a witness hyperedge $\vec{z_j}$ such that every vertex in $\vec{z_i}$ either does not appear in any other hyperedge, or it appears in $\vec{z_j}$.

If an edge elimination leads to a vertex no longer being covered by any hyperedge, the vertex will be deleted as well. If no vertices and edges are left in $\mathcal{H}$ at the end, then the query is acyclic. According to Tarjan et al. [TY84], the GYO reduction satisfies

the Church-Rosser property. This means that the algorithm always ends with the same result, regardless of the order in which the two rules are applied.

We now show the GYO reduction with example join query Q_1 (based on [Pic21]) that is defined as follows:

$$out(x_1, x_2, x_3, x_4, x_5, x_6) :- R_1(x_1, x_2, x_3), R_2(x_2, x_3), R_3(x_3), R_4(x_2, x_4, x_3), R_5(x_5, x_6).$$

The hypergraph $\mathcal{H}_{Q_1} = (V, E)$ of query Q_1 has vertices $V = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and hyperedges $E = \{\{x_1, x_2, x_3\}, \{x_2, x_3\}, \{x_3\}, \{x_2, x_4, x_3\}, \{x_5, x_6\}\}$.

1. We start the GYO reduction with a full edge set, i.e. $E' = E$.
2. We apply the first rule to remove edge $\{x_5, x_6\}$ from E' since its vertices do not appear in any other hyperedge in E'. Thus, $E' = \{\{x_1, x_2, x_3\}, \{x_2, x_3\}, \{x_3\}, \{x_2, x_4, x_3\}\}$.
3. We apply the second rule to remove edge $\{x_3\}$ from E' since it is contained in edge $\{x_1, x_2, x_3\}$. We now have $E' = \{\{x_1, x_2, x_3\}, \{x_2, x_3\}, \{x_2, x_4, x_3\}\}$.
4. We apply the second rule to remove edge $\{x_2, x_4, x_3\}$ since x_4 does not appear in any other hyperedge and $\{x_2, x_3\}$ is contained in edge $\{x_1, x_2, x_3\}$. Therefore, $E' = \{\{x_1, x_2, x_3\}, \{x_2, x_3\}\}$.
5. We apply the second rule to remove edge $\{x_1, x_2, x_3\}$ since x_1 does not appear in any other hyperedge and $\{x_2, x_3\}$ is contained in edge $\{x_2, x_3\}$. Consequently, $E' = \{\{x_2, x_3\}\}$.
6. We apply the first rule to remove edge $\{x_2, x_3\}$ from E' since its vertices do not appear in any other hyperedge in E'. Thus, $E' = \emptyset$.
7. The algorithm terminates now as no rule can be applied. Since $E' = \emptyset$, we know that Q_1 is acyclic.

We can then derive a join tree from the GYO elimination sequence as follows [Yan81]:

1. If the hyperedge associated with atom R_i has a witness hyperedge associated with atom R_j, R_i is made child node of R_j in the join tree.
2. Otherwise, R_j is simply added as a node in the forest.
3. The resulting join forest is merged arbitrarily to obtain a join tree.

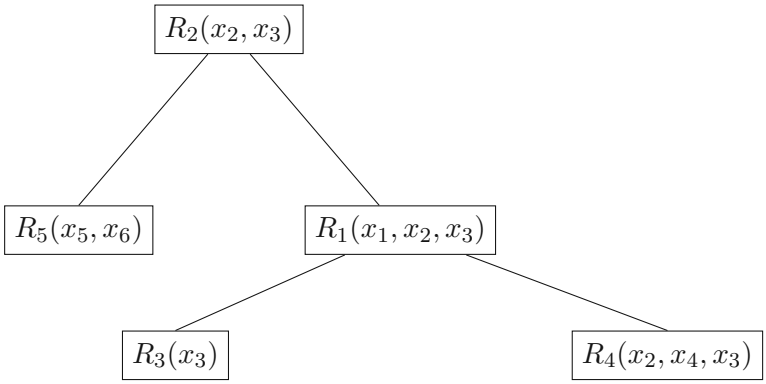


Figure 3.1: Join tree T_{Q_1} for query Q_1 (based on [Pic21])

Figure 3.1 shows a join tree T_{Q_1} that has been derived from the prior example query Q_1 and the application of the GYO reduction.

It would be desirable if joins were monotonic, i.e. no tuple is removed during any of the joins. Monotonicity guarantees that the intermediate results generated during the joins are a subset of the final result, with no additional tuples that are not included in the final result. In general, monotonicity is not satisfied, as the following counter-example by Beeri et al. [BFMY83] shows. They considered a scenario where four relations, r_1 r_2, r_3 and r_4, are joined together. The first join operation between r_1 and r_2 results in an intermediate result size of thousand tuples. Joining this intermediate result with r_3 leads to a set of one million tuples. Finally, the last join with r_4 yields a result containing just ten tuples. In this case, the monotonicity property is clearly violated, as tuples are removed by join operations. To obtain monotonicity, we need the join of k relations $R^D = \bowtie_{1 \leq i \leq k} R_i^D$ to be *globally consistent* over a database D, i.e. it holds that $R_j^D = \pi_{R_j}(R^D)$ for each relation R_j. In case of acyclic queries, global consistency is achieved if *pairwise consistency* is satisfied. Pairwise consistency holds if every pair R_i, R_j with $i \neq j$ of relations is *consistent* over D, meaning that their projections on shared variables are equal, i.e. $\pi_{z_i \cap z_j} R_i^D = \pi_{z_i \cap z_j} R_j^D$. In the following, we discuss an algorithm that performs semi-joins along a join tree to establish pairwise consistency, enabling a monotonic join to be performed between the relations involved in the acyclic query.

3.2 Yannakakis algorithm

Yannakakis [Yan81] has proposed an algorithm for acyclic queries that can (1) decide in polynomial time whether the result of a query over a database D is empty, and (2) compute the result set of a query over D in output polynomial time. The algorithm relies on a join tree $T = (V, E)$ and a database D. It needs either one (for boolean queries) or three tree traversals with alternating sequence.

1. Each relation in the database R_i^D is associated with its corresponding node $R_i \in V$.

2. A bottom-up traversal is performed. For each child node $R_j(\vec{z_j})$ of $R_i(\vec{z_i})$ having at least one common variable, i.e. $\vec{z_j} \cap \vec{z_i} \neq \varnothing$, a semi-join is computed such that all tuples from the parent relation (R_i^D) are removed for which there is no join partner in the child relation (R_j^D). If the emerging relation at the root node is subsequently empty, the result of the query over D is also empty.

3. If it is not empty, we continue with the second tree traversal, which is done from top to bottom. For this, a semi-join is computed such that all tuples from the child relation (R_j^D) are removed for which there is no join partner in the parent relation (R_i^D).

4. In the third pass, we conduct a bottom-up traversal again. This time, however, we perform regular joins so that the relation at the root node contains the result of the query over D after all. In case two nodes $R_j(\vec{z_j})$ and $R_i(\vec{z_i})$ do not share any common variable, a cross join is performed.

The use of the two semi-join passes prevents the exponential growth of intermediate results. Semi-joins are an efficient form of selection, keeping only those tuples for which there is a matching partner. When performing the third pass, which prunes the tree from the leaves to the root, we are guaranteed that everything we have as an intermediate result is also included in the final join result. It is only extended by additional attributes. This is made possible by the two semi-join passes carrying out a "full reduction" of a set of relations $\{R_1, \ldots, R_k\}$ over D that correspond to the atoms in the acyclic query. The set is considered to be "fully" reduced, if the relations are *globally consistent* over D.

We now illustrate the application of the Yannakakis algorithm using the query Q_1 (defined in Section 3.1) along some example data for the involved relations. Figure 3.2 shows the application of the first bottom-up traversal. Each grey tuple has been removed by the application of some semi-join. The algorithm has first removed the tuples from relation $R_1(x_1, x_2, x_3)$ due to them having no join partner in $R_3(x_3)$ or $R_4(x_2, x_4, x_3)$. No semi-join was performed between $R_2(x_2, x_3)$ and $R_5(x_5, x_6)$ since these relations have no variable in common. As a final step in this traversal, the tuple from $R_2(x_2, x_3)$ got removed from $R_1(x_1, x_2, x_3)$ because there was no matching tuple in $R_1(x_1, x_2, x_3)$. Since the result at the root node is non-empty at this point, we continue with the top-down traversal shown in Figure 3.3. The tuples which got removed with the prior traversal are now shown in light-grey for reference. Only one tuple is removed with the top-down traversal, i.e. value b_3 from relation $R_3(x_3)$ is removed by a semi-join because there is no longer a join partner in $R_1(x_1, x_2, x_3)$. All remaining tuples will be part of the final join result (given in Table 3.1) which is computed by the second bottom-up traversal.

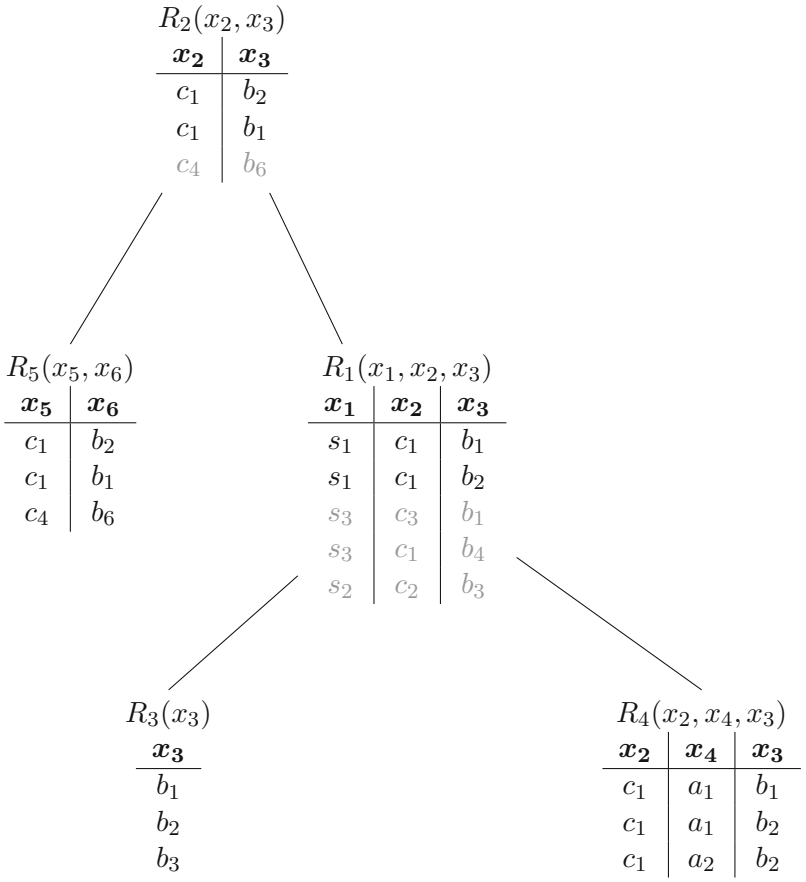


Figure 3.2: Apply bottom-up traversal with Yannakakis algorithm for query Q_1 (based on [Pic21])

3.3 Generalization of acyclic queries

There has been interest among researchers in extending the benefits of acyclic queries to a larger class of queries, since a significant proportion of queries encountered in practice are not acyclic, but nearly acyclic. As a result, since the 1990s, efforts have been made in research to weaken the notion of acyclicity by defining concepts for near acyclicity, along with the development of algorithms that can efficiently evaluate such queries. To specify the degree of acyclicity for a query Q and its associated hypergraph $\mathcal{H}_Q$, different notions of width have been proposed over time. According to Gottlob et al., for a fixed constant $k \geq 1$, a good generalization for acyclicity should satisfy the following three fundamental criteria [GGLS16].

1. **Generalization of acyclicity**: The class of queries having width bounded by k should include all acyclic queries.

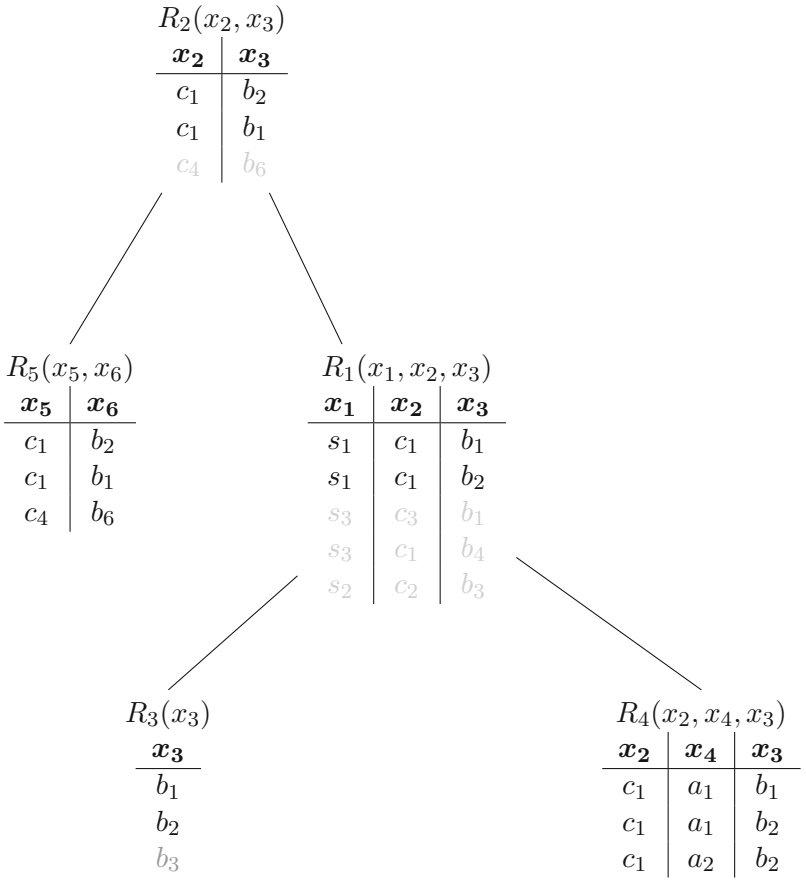


Figure 3.3: Apply top-down traversal with Yannakakis algorithm for query Q_1 (based on [Pic21])

2. **Tractable recognizability**: An algorithm exists which decides if the width of a given query is $\leq k$ for fixed k in polynomial time.

3. **Tractable query-answering**: There should be an algorithm that can evaluate a query in polynomial time (in the combined size of the input and output for non-boolean CQs).

In [GGLS16], Gottlob et al. have reviewed various decomposition methods and associated width measures found in the literature. They have also analysed whether these methods fulfil the three previously defined criteria. Moreover, they have outlined how such a decomposition-based method can be used to more efficiently evaluate a broader set of queries that may contain "mild cyclicity". The remainder of this section will therefore mainly rely on this work.

x_1	x_2	x_3	x_4	x_5	x_6
s_1	c_1	b_2	a_1	c_1	b_2
s_1	c_1	b_2	a_1	c_1	b_1
s_1	c_1	b_2	a_1	c_4	b_6
s_1	c_1	b_2	a_2	c_1	b_2
s_1	c_1	b_2	a_2	c_1	b_1
s_1	c_1	b_2	a_2	c_4	b_6
s_1	c_1	b_1	a_1	c_1	b_2
s_1	c_1	b_1	a_1	c_1	b_1
s_1	c_1	b_1	a_1	c_4	b_6

Table 3.1: Result after third pass of Yannakakis algorithm on query Q_1 (based on [Pic21])

Tree Decomposition

Tree decomposition and its associated width measure *treewidth* [RS84, RS86] are well-known concepts from graph theory that have originally been applied for the generalization of graph acyclicity. However, we can also utilize these concepts in the context of hypergraphs and thus for the evaluation of conjunctive queries. A tree decomposition of a hypergraph $\mathcal{H} = (V, E)$ consists of a pair $\langle T, \chi \rangle$, in which $T = (V', F)$ is the tree and χ is a function that assigns to each node $p \in V'$ a set of vertices $\chi(p) \subseteq V$, which is also referred to as the *bag* of p. Furthermore, the following three conditions must be satisfied for a valid tree decomposition.

1. **Vertex covering**: For every vertex v from the hypergraph $\mathcal{H}$ there is at least one tree node p containing the vertex in its bag $\chi(p)$. Formally, there must be a $p \in V'$ such that $v \in \chi(p)$, for every vertex $v \in V$.
2. **Hyperedge covering**: For every hyperedge $e \in E$, there must be a tree node $p \in V'$, whose associated bag $\chi(p)$ fully covers it, i.e. $e \subseteq \chi(p)$.
3. **Connected subtrees**: For each vertex $v \in V$, the set of tree nodes whose associated bag contains v, i.e. $\{p \mid p \in V' \wedge v \in \chi(p)\}$, induces a connected subtree within T.

The *width* of a tree decomposition $\langle T, \chi \rangle$ is defined as the cardinality of the largest bag within T minus one, i.e. $max_{p \in V'} (|\chi(p)| - 1)$. The degree of acyclicity of a hypergraph $\mathcal{H}$ is denoted by the *treewidth* $tw(\mathcal{H})$, which corresponds to the minimum width of all tree decompositions of $\mathcal{H}$. Accordingly, the *treewidth* $tw(Q)$ of a conjunctive query Q is defined by the treewidth $tw(\mathcal{H}_Q)$ of its associated hypergraph $\mathcal{H}_Q$.

We now consider the cyclic query Q_2 (taken from [Got20]), which can be defined as follows.

$$out() : -a(S, X, X', C, F), b(S, Y, Y', C', F'), c(C, C', Z), d(X, Z), e(Y, Z), f(F, F', Z'),$$
$$g(X', Z'), h(Y', Z'), j(J, X, Y, X', Y'), p(B, X', F), q(B', X', F).$$

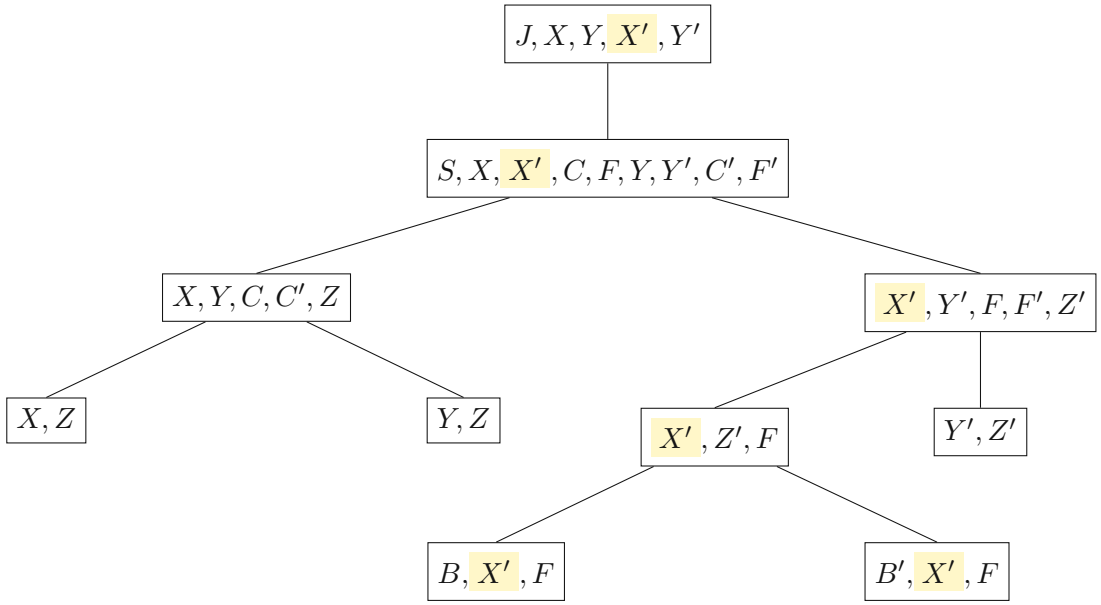


Figure 3.4: Tree decomposition for query Q_2 (based on [Got20])

An example tree decomposition for query Q_2 is given in Figure 3.4. The tree decomposition has a width of 8, since the largest bag $\{S, X, X', C, F, Y, Y', C', F'\}$ contains 9 variables. The "connected subtrees" property is obviously satisfied for each variable. We demonstrate this with the variable X', which we have highlighted with a yellow background.

Having defined *tree decomposition* and the associated *treewidth* measure, we will now examine the three criteria that make up a good generalization for acyclicity of hypergraphs and check whether all of them are satisfied.

1. **Generalization of acyclicity**: The notion of *treewidth* only generalizes graph acyclicity, i.e. a graph G is acyclic if and only if $tw(\mathcal{G})$ is 1. However, this does not hold for hypergraphs. For example, the class of single-atom conjunctive queries has unbounded tree-width, but is acyclic. Consequently, *treewidth* is not a generalization for acyclicity of hypergraphs.

2. **Tractable recognizability**: In general, the computation of the treewidth $tw(\mathcal{H})$ is NP-hard. However, for fixed k, we can check if $tw(\mathcal{H})$ is $\leq k$, and in this case compute a tree decomposition of optimal width using a linear-time algorithm that operates in logarithmic space. As a result, tree decompositions satisfy this condition.

3. **Tractable query-answering**: Conjunctive queries Q with $tw(Q)$ of k can be evaluated in time $O(m' * D^{k+1} * log\ D)$ over a database, where m' refers to the

number of tree nodes in T, and D represents the number of unique values in the DB. The tree decomposition method thus satisfies this condition.

Generalized Hypertree Decomposition

Since tree decomposition is not a good generalization for acyclicity, an extension called *Generalized Hypertree Decomposition* (or *GHD* for short) has been developed to overcome this limitation. The extension has the form of an additional labelling λ which assigns to every tree node $p \in V'$ a set $\lambda(p)$ of atoms (or hyperedges from $\mathcal{H}$) such that the bag $\chi(p)$ of variables (or vertices from $\mathcal{H}$) is covered by it. This is similar to join trees, but in case of a GHD, more than one atom per node is allowed. Formally, a *generalized hypertree decomposition* of a hypergraph $\mathcal{H}$ is defined as a triple $HD = \langle T, \chi, \lambda \rangle$, referred to as *hypertree* for $\mathcal{H}$, where $\langle T, \chi \rangle$ denotes a tree decomposition of $\mathcal{H}$, and λ is a function which labels vertices of T with sets of hyperedges of $\mathcal{H}$, such that the following condition holds. For every tree node $p \in V'$, $\chi(p) \subseteq \bigcup_{h \in \lambda(p)} h$. Intuitively, this means that for all tree nodes $p \in V'$, every variable (or vertex from $\mathcal{H}$) assigned to p with $\chi(p)$ must be covered by at least one hyperedge associated to it with $\lambda(p)$.

The *width* of a given GHD is defined as the maximum number of hyperedges used to cover a single set, i.e. $max_{p \in V'}(|\lambda(p)|)$. The degree of acyclicity of a hypergraph $\mathcal{H}$ is defined similarly as before with $ghw(\mathcal{H})$, which corresponds to the minimum width of all GHDs of $\mathcal{H}$. Thus, the goal in computing a GHD is to find a GHD that contains as few atoms (or hyperedges) per set as possible.

We will now revisit the three criteria that make a good generalization for the acyclicity of hypergraphs in order to check their fulfilment.

1. **Generalization of acyclicity**: By the definition of *ghw*, we have $ghw(\mathcal{H}) = 1$ for an acyclic hypergraph $\mathcal{H}$. This is the case because a hypergraph $\mathcal{H}$ is acyclic if and only if a join tree exists. Since we assign exactly one atom to each tree node of a join tree, we obtain $ghw(\mathcal{H}) = 1$. Consequently, GHDs satisfy this criterion.
2. **Tractable recognizability**: Gottlob et al. [GMS09, GLPR21] have found that it is NP-hard to decide whether the hypergraph $\mathcal{H}_Q$ of a query Q has $ghw(\mathcal{H}_Q) \leq k$ for a $k \geq 2$. Thus, this criterion is not fulfilled.
3. **Tractable query-answering**: Conjunctive queries Q with $ghw(Q) = k$ can be evaluated in polynomial time, provided that a generalized hypertree decomposition of this width is given. The same time complexity applies as in hypertree decompositions, which are considered next. This criterion is therefore satisfied for GHDs.

Hypertree Decomposition

To solve the problem of *tractable recognizability*, an additional constraint has been invented for GHDs, called *Descendant Condition* or *Special Condition*. Formally, the

condition is satisfied if $h \cap \chi(T_p) \subseteq \chi(p)$ holds for every tree node p of V' and hyperedge $h \in \lambda(p)$, where T_p refers to the subtree of T that is rooted at p, and $\chi(T_p)$ denotes the set of variables contained in the χ labelling of the subtree T_p. The notion of *hypertree decomposition* refers to a GHD that satisfies this condition.

Intuitively, the special condition is being applied during the top-down construction of a width k-GHD for a hypergraph $\mathcal{H}$ with the intent to simplify the computation. For each inner node in the hypertree decomposition, we guess all possible λ labels for its child nodes. Since each label can consist of up to k atoms, there are polynomially many possibilities. However, specifying the χ-label for a child node is more complex, since there are exponentially many possibilities for the χ-bag of a node, if the edges have arbitrary size. To restrict the choice of possible child bags to a polynomial number of alternatives, the special condition is used. Each of these alternatives can be represented in logarithmic space. When an atom appears for the first time, it must appear fully, i.e., all variables must be part of the χ label for the node. For descendants, this atom may then appear partially, meaning that not all variables are part of the bag $\chi(p)$. If a variable is removed from the bag $\chi(p)$ of some node, it may not appear in any descendant nodes. Moreover, all nodes that occur in the hyperedges of $\lambda(p)$ but are not included in $\chi(p)$ are not effective for p and therefore do not count towards the connectedness condition.

The width of a *hypertree decomposition* and $hw(\mathcal{H})$ is defined like its corresponding notions for *generalized hypertree decomposition*.

We now revisit the previously defined cyclic query Q_2. An example hypertree decomposition HD_{Q_2} for the query Q_2 is shown in Figure 3.5. If we compare it with the tree decomposition of Q_2 shown in Figure 3.4, we can see that for each tree node in HD_{Q_2}, the set of atoms covers the set of variables of the corresponding node in the tree decomposition. Since we have at most two atoms per tree node, the width of the hypertree decomposition is 2. To satisfy the connectedness condition, a projection of an atom is specified for some tree nodes. The removed variables are marked with the character _. These projections can only be made for an atom within a given tree node if the "full atom" occurs at least once in a predecessor node. One example is the atom $\boldsymbol{j}$, where three missing variables are highlighted in yellow in a tree node. The corresponding complete atom appears in the root node. Furthermore, any variable that has disappeared in a tree node cannot reappear in any descendant node (highlighted in blue) due to the descendant condition.

We will now show that all three criteria that constitute a good generalization for the acyclicity of hypergraphs are satisfied for hypertree decompositions.

1. **Generalization of acyclicity**: By the definition of hw, we have $hw(\mathcal{H}) = 1$ for an acyclic hypergraph $\mathcal{H}$. This is the case because a hypergraph $\mathcal{H}$ is acyclic if and only if a join tree exists. Since we assign exactly one atom to every tree node of a join tree, we obtain $hw(\mathcal{H}) = 1$. Thus, this criterion is fulfilled.

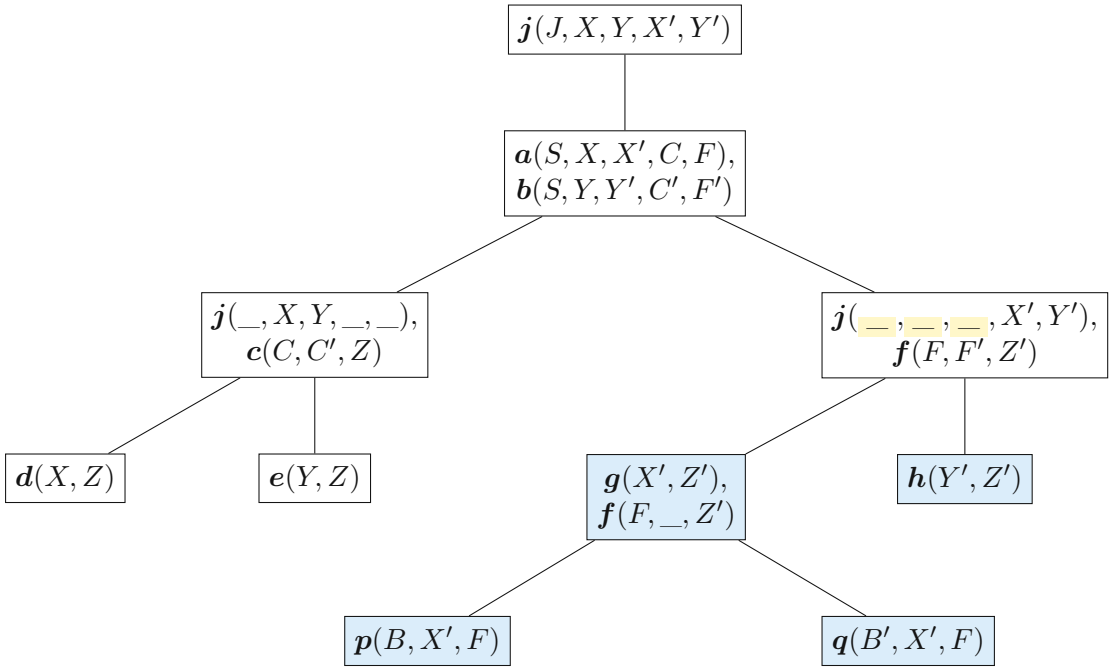


Figure 3.5: Hypertree decomposition HD_{Q_2} (based on [Got20])

2. **Tractable recognizability**: Due to the descendant condition, a hypertree decomposition with $hw(\mathcal{H}_Q) = k$ can be computed for the hypergraph $\mathcal{H}_Q$ of a query Q in polynomial time using an alternating logspace procedure. The best currently known upper bound on the time complexity of a deterministic realization of the alternating procedure is $O(m^{2k}v^2)$, where m denotes the number of hyperedges and v the number of vertices in $\mathcal{H}_Q$. It is important to note that k is involved as a factor in the exponent of the polynomial. According to Gottlob et al., this is not really a problem in the context of query evaluation in practice for three reasons. First, a vast majority of conjunctive queries have very low $hw(\mathcal{H}_Q)$ of up to 3. Second, they reason that the number of atoms in conjunctive queries hardly reaches more than 50. Third, they argue that it might be worthwhile to spend a little more computational effort to get a very good decomposition, because in practice queries are re-executed very often in a database system, and therefore we could benefit numerous times when executing a highly-optimised query.

3. **Tractable query-answering**: Conjunctive queries Q with $hw(Q) = k$ can be evaluated in polynomial time with an upper bound of $O(v * (r^k + s) * log(r + s))$ over a database, where r corresponds to the size of the largest relation involved in the query, v denotes the number of variables in Q, and s is the number of output tuples. Hence, this criterion is fulfilled.

Evaluation of queries using hypertree decomposition

While there are several approaches on how we can use a hypertree decomposition to evaluate a query, we will focus on the most natural one here.

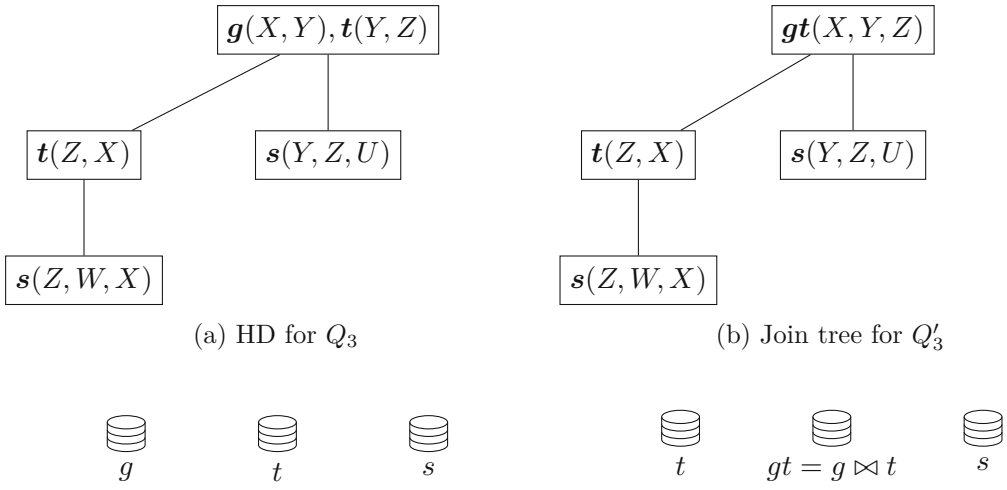
Let Q be a conjunctive query with an associated hypergraph $\mathcal{H}_Q$ that we want to evaluate over a database instance DB. Furthermore, let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition for $\mathcal{H}_Q$ with $hw(\mathcal{H}_Q) = k$. The basic idea is to transform the query Q using HD into an acyclic query Q' with associated join tree $T_{Q'}$ so that we can apply the Yannakakis algorithm as discussed in Section 3.2. First, for each tree node $p \in V'$, we compute a fresh atom, which then replaces all atoms assigned to p as hyperedges in $\lambda(p)$. A fresh atom is obtained by (1) joining all the relations associated with the atoms currently assigned to p with $\lambda(p)$, and (2) projecting the result of the join on the set of variables assigned to p with $\chi(p)$. As a result of these computations, we get a new database instance, which we refer to as DB'. Due to the hypertree width k, at most k relations are to be joined per tree node p, which can be done in polynomial time. Since χ encodes a tree decomposition, we then have a join tree $T_{Q'}$, whose conjunction of atoms would represent an acyclic conjunctive query Q'. The queries Q and Q' are equivalent in the sense that the execution of Q' over DB' yields identical results to that of Q over DB. Second, we apply the Yannakakis algorithm on $T_{Q'}$ and the associated relations in DB'.

We now outline the complexity of evaluating query Q using the approach described earlier, as reported in [GGLS16]. The first step can be done in polynomial time bounded by $O(m * r^k)$, where m is the number of vertices in the hypertree, and r is the size of the largest relation in the database instance DB. Let DB' have at most r^k tuples in the largest relation after performing the first step, and let $r' \leq r^k$ be the actual number of tuples in the largest relation of DB'. The total complexity of evaluating Q is then obtained by adding the cost of evaluating the acyclic instance Q', which dominates the total cost. The algorithm has, in the worst case, an upper bound of $O(m * (r' + s) * log(r' + s))$ for the runtime and $O(m * (r' + s))$ for the space, where s denotes the size of the output.

We now illustrate this approach for query evaluation with cyclic query Q_3 (taken from [Got20]), which can be defined as follows.

$$out() : -s(Y, Z, U), g(X, Y), t(Z, X), s(Z, W, X), t(Y, Z).$$

First, hypertree decomposition HD_{Q_3} is computed (see Figure 3.6a). Then, we use HD_{Q_3} and the relations in database instance DB (see Figure 3.6c) to derive a join tree (see Figure 3.6b), which corresponds to the acyclic conjunctive query Q'_3, and database instance DB' (see Figure 3.6d). As part of this step, the two atoms $\boldsymbol{g}(X, Y)$ and $\boldsymbol{t}(Y, Z)$ are replaced with fresh atom $\boldsymbol{gt}(X, Y, Z)$ at the root node of the tree. Beyond that, relation $\boldsymbol{gt} = g \bowtie t$ is computed, which results in the creation of database instance DB'. Finally, the Yannakakis algorithm can be applied using the join tree Q'_3 and database instance DB'.



(a) HD for Q_3

(b) Join tree for Q_3'

(c) Relations of DB

(d) Relations of DB'

Figure 3.6: Transformation of query Q_3 into acyclic query Q_3' (based on [Got20])

CHAPTER 4

Worst-case optimal joins

In this chapter, we will discuss worst-case optimal joins as a solution to the problem of exploding intermediate results in join queries. We shall use the triangle query, which is the most basic form of a cyclic join query, as an example.

Suppose we have relations $R(a, b), S(b, c), T(a, c)$, with each having a cardinality of n. When performing two-way joins, i.e. one relation is joined with another relation and then the result is joined with the third relation, the worst-case intermediate result size is in $\Omega(n^2)$ if the data is highly skewed, regardless of how we choose the order of the joins. However, AGM (Atserias, Grohe, Marx) [AGM13] have shown that there is an upper bound for a join query with regard to the result size, depending on the size of the input. As a result, we know that the size of the final result is bounded by $O(n * \sqrt{n})$ for the triangle query. There is obviously a blow-up in between, which produces unnecessarily large intermediate results.

Worst-case optimal join algorithms have been proposed to solve this problem. These are characterized by the fact that their time complexity is always within the AGM [AGM13] bound, i.e. it remains within the largest possible output size. This has been an active area of research over the last 5 years. Hence, there are now several such methods. We limit ourselves here to the original approach, called *Leapfrog-Triejoin*, and another one that excels in its simplicity.

In summary, this chapter will formally define the AGM bound and present two algorithms for worst-case optimal joins.

4.1 AGM bound

We will first introduce some key terms that will be necessary later on. An edge cover of a hypergraph $\mathcal{H} = (V, E)$ is a subset $C \subseteq E$ of the edges of $\mathcal{H}$ such that every vertex of $\mathcal{H}$ is covered by at least one hyperedge in C. The *edge cover number* $\rho(\mathcal{H})$ of $\mathcal{H}$ corresponds

to the minimum size of C among all edge covers of $\mathcal{H}$. A fractional edge cover of a hypergraph $\mathcal{H} = (V, E)$ assigns a weight $\psi(e)$ in $[0, \infty)$ to every hyperedge $e \in E$ such that every vertex $v \in V$ has a total weight of at least 1, i.e. $\sum_{e \in E, v \in e} \psi(e) \geq 1$ for every $v \in V$. The *fractional edge cover number* $\rho^*(\mathcal{H})$ of $\mathcal{H}$ corresponds to the minimum of the total weights among all fractional edge covers of $\mathcal{H}$. An optimal fractional edge cover can be computed in polynomial time using a linear program that minimizes the cost $\rho^*(\mathcal{H})$. For a join query Q, the (fractional) edge cover number is defined by its associated hypergraph, i.e. $\rho(Q) = \rho(\mathcal{H}_Q)$ respectively $\rho^*(Q) = \rho^*(\mathcal{H}_Q)$. Let's consider the triangle query again as an example. Figure 4.1 shows a visualization of an edge cover and a fractional edge cover for it. A valid edge cover would be any subset of edges that is composed of at least two of the three hyperedges. Hence, we have $\rho(Q) = 2$. We can construct a fractional edge cover $x_R = x_S = x_T = \frac{1}{2}$. The total of these weights are the minimum among all fractional edge covers of $\mathcal{H}_Q$. Hence, $\rho^*(Q) = \frac{3}{2}$ [GM06a, AGM13].

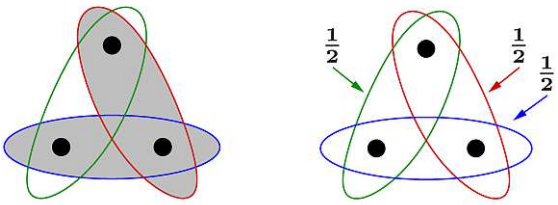


Figure 4.1: Visualization of an edge cover and a fractional edge cover for the triangle query (based on [GM06b])

Atserias et al. [AGM13] observed that a fractional edge cover can be used for the definition of a lower bound for the worst-case output size of a join query. Let Q be a join query consisting of the join of n relations $R_1, R_2, \ldots, R_n$, and let D be a database instance. Furthermore, let $x_1, x_2, \ldots, x_n$ be a fractional edge cover for Q, such that x_i is the weight of the hyperedge associated with R_i for $1 \leq i \leq n$. Then, we have

$$|Q(D)| \leq \prod_{i=1}^{n} |R_i|^{x_i}. \tag{4.1}$$

It follows that the size of the output of the triangle query is bounded by $|R|^{\frac{1}{2}} * |S|^{\frac{1}{2}} * |T|^{\frac{1}{2}}$. If we assume equal relation sizes, i.e. $|R| = |S| = |T| = n$, then we have $n^{\frac{3}{2}}$ or $n * \sqrt{n}$.

4.2 Leapfrog-Triejoin algorithm

Leapfrog Triejoin [Vel14] is the first worst-case optimal join algorithm that has been used in a commercial database system. It achieves worst-case optimal running time (up to a log factor) for some classes of database instances. Instead of performing multiple two-way joins in a specific order – with the need for potentially big intermediate results – like a traditional DBMS, the algorithm from company LogicBlox can join multiple relations simultaneously by performing a backtracking search based on the variables that are part of the join. The algorithm assumes that the relations to be joined are sorted. It relies on linear iterators for accessing the individual relations. Each iterator is required to take at most $log(N)$ time – with N being the number of tuples in the corresponding relation – for getting the next value or for seeking a given value from/in a relation for a specific variable. Beyond that, an iterator must take not more than $O(1 + log(N/m))$ time, if m consecutive values are retrieved from a specific variable from a relation. Ngo et al. [NRR14] mentioned in their survey that this can be achieved either using a balanced tree structure such as a B-tree, or a hash-based indexing approach.

Unary joins, referred to as *leapfrog joins* by Veldhuizen, form the basic building block. A leapfrog join is based on a sort-merge join and can be used to simultaneously join a set of unary relations $A_1(x), \ldots, A_k(x)$. Each leapfrog join keeps a list of pointers to iterators, one for each relation, and stores the smallest and largest values at which the iterators are currently positioned. It repeatedly moves the iterator with the lowest value to a value that is equal or greater than the highest value from all iterators. This is done until all iterators have the same value – yielding a new output value. When a leapfrog join iterator is constructed, the *leapfrog-init* (Algorithm 4.1) method is first called, which initializes its state and finds the first result. The method *leapfrog-search* (Algorithm 4.2) finds the next value in the intersection $A_1(x) \cap \cdots \cap A_k(x)$. After we have obtained the first value from the intersection, we can obtain subsequent ones by calling the *leapfrog-next* (Algorithm 4.3) method. In order to seek for a value from the intersection that is greater than or equal to a given value, we can utilize the *leapfrog-seek* (Algorithm 4.4) method.

Algorithm 4.1: leapfrog-init() [Vel14]

```
1 if any iterator has atEnd() true then
2 |   atEnd := true;
3 else
4 |   atEnd := false;
5 |   sort the array Iter[0..k - 1] by keys at which the iterators are positioned;
6 |   p := 0;
7 |   leapfrog-search()
```

Algorithm 4.2: leapfrog-search() [Vel14]

```
1  x' := Iter[(p - 1) mod k].key(); // Max key of any iter
2  while true do
3      x := Iter[p].key(); // Least key of any iter
4      if x = x' then
5          key := x; // All iters at same key
6          return;
7      else
8          Iter[p].seek(x');
9          if Iter[p].atEnd() then
10             atEnd := true;
11             return;
12         else
13             x' := Iter[p].key();
14             p := p + 1 mod k;
15 end
```

Algorithm 4.3: leapfrog-next() [Vel14]

```
1  Iter[p].next();
2  if Iter[p].atEnd() then
3      atEnd := true;
4  else
5      p := p + 1 mod k;
6      leapfrog-search();
```

Algorithm 4.4: leapfrog-seek(int seekKey) [Vel14]

```
1  Iter[p].seek(seekKey);
2  if Iter[p].atEnd() then
3      atEnd := true;
4  else
5      p := p + 1 mod k;
6      leapfrog-search();
```

Figure 4.2 shows a leapfrog join of the three unary relations A, B, C. The iterators are

initially positioned at values 0, 0 and 2 respectively. The execution of *seek*(2) for the iterator of relation A moves the iterator to value 3. After that, a *seek*(3) operation is carried out on the iterator of relation B, which results in a move to the value 6. Subsequently, the iterator of C seeks for value 6, which results in a move to value 8. After that, the iterators of relations A and B are moved to 8 as well with a *seek*(8) operation. Then, all iterators have the same value at their current position. Thus, 8 is the first value being returned by the leapfrog join.

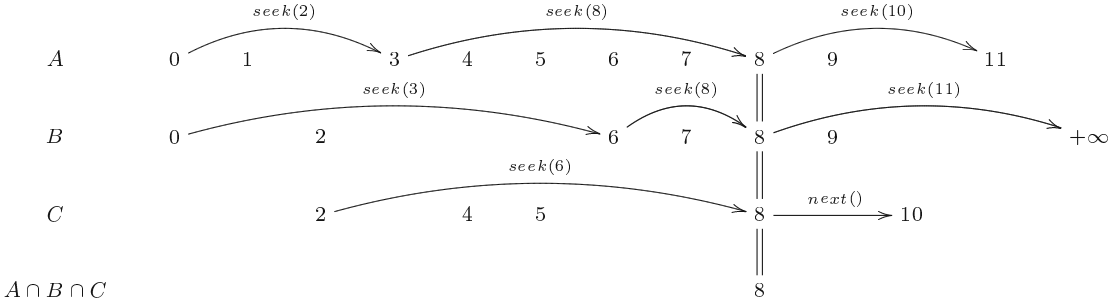


Figure 4.2: Leapfrog join of unary relations A, B and C [Vel14]

In some scenarios, a leapfrog join can significantly outperform a two-way join. For example, consider the join of unary relations $A = \{0, \ldots, 2n-1\}$, $B = \{n, \ldots, 3n-1\}$ and $C = \{0, \ldots, n-1, 2n, \ldots, 3n-1\}$. In this case, each pairwise join would lead to n tuples being included in the intermediate result, while the final result is empty. Leapfrog join is able to compute this in $O(1)$ steps.

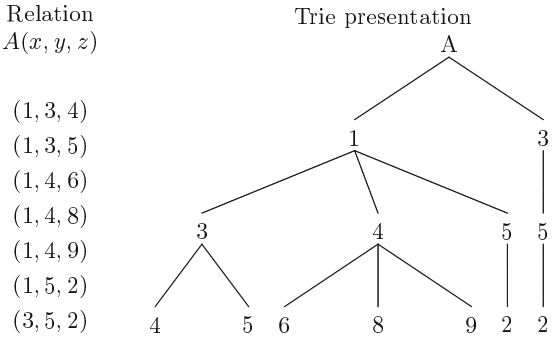


Figure 4.3: Trie representation of relation $A(x, y, z)$ [Vel14]

The trie iterator interface extends this concept to relations with arity > 1. As can be seen in Figure 4.3, each tuple (x_i, y_i, z_i) from relation $A(x, y, z)$ should be presented as if it would be stored in a trie such that there is a unique path from the root to a leaf node. Upon initialization, the trie iterator is positioned at the root node, i.e. at A. It keeps track of the current depth within the trie. We can move to the first value at the next depth with the *open* (Algorithm 4.5) method. The *up* (Algorithm 4.6) method can be used to return to the parent value at the prior depth. After the *open* method has been called at some node n, the *next*, *seek* and *atEnd* methods will operate on the children of

n. For example, if we invoke the *open* method three times after being at node A, the iterator would be positioned at node $[1, 3, 4]$. Calling the *next* method would then move the iterator to node $[1, 3, 5]$. Another *next* method call would result in the atEnd method returning true. We could then execute the sequence of method calls *up*, *next* and *open* to reposition the iterator to leave node $[1, 4, 6]$.

Algorithm 4.5: triejoin-open() [Vel14]

```
1 depth := depth + 1; // Advance to next var
2 for each iter in leapfrog join at current depth do
3 |   iter.open();
4 end
5 call leapfrog-init() for leapfrog join at current depth
```

Algorithm 4.6: triejoin-up() [Vel14]

```
1 for each iter in leapfrog join at current depth do
2 |   iter.up();
3 end
4 depth := depth - 1 ; // Backtrack to previous var
```

We now consider the triangle join query $R(a, b) \bowtie S(b, c) \bowtie T(a, c)$ as an example. The algorithm creates a *leapfrog* instance for every variable, with each of them having pointers to the iterators of the relations which the variable occurs in. Beyond that, the optimizer has to choose a variable ordering, i.e. a permutation of the variables which appear in the join. Assume we choose variable ordering $[a, b, c]$. The join utilizing a "backtracking search" is then performed as follows. First, a unary join is performed for the variable a with the projections $\pi_a R$ and $\pi_a T$. As soon as this emits a value a_i, a unary join is performed for variable b between $\pi_b(\sigma_{a=a_i} R)$ and $\pi_b S$. For each b_i, a unary join is then performed for variable c between $\pi_c(\sigma_{b=b_i} S)$ and $\pi_c(\sigma_{a=a_i} T)$. If all values c_i are exhausted, we go to the previous level and obtain the next value for the previous variable.

4.3 Partitioning-based algorithm

A well-known worst-case optimal join (WCOJ) algorithm builds upon fundamental ideas outlined in [NRR14]. On a high-level the algorithm partitions some relation based on the "light hitters" and "heavy hitters" values of some join attribute. The goal is to avoid the explosion of the first intermediate result for each of the partitions, i.e. avoid the worst-case cardinality $\Omega(n^2)$ of a regular two-way join. This is achieved (1) by selecting a good partitioning threshold, and (2) by choosing a different sequence of joins for the two partitions depending on whether the join attribute chosen for the partitioning exists in the other relations. In the end, both results are combined.

We will now illustrate this algorithm with the triangle join query $Q = R(X,Y) \bowtie S(Y,Z) \bowtie T(X,Z)$. Assume that the cardinality of every relation is equal, i.e. $|R| = |S| = |T| = n$. As can be seen in Figure 4.4, we select relation R to be split into relations R^+ and R^- based on attribute X. We assign a tuple of the relation R to the "heavy hitters" partition R^+ if its value in attribute X occurs more than $\sqrt{n}$ times within the same attribute in the entire relation R, or to the "light hitters" partition R^- otherwise. We observe that due to our partitioning, there are at most $\sqrt{n}$ different X values in R^+, because otherwise there would be more than $\sqrt{n} * \sqrt{n} = n$ tuples which cannot be the case since $|R| = n$. As a result, we join R^+ with S because for every tuple from S there are at most $\sqrt{n}$ X values. This yields an intermediate result size bound of $O(n * \sqrt{n})$. This corresponds to the worst-case cardinality of a join, i.e. a Cartesian product, where every tuple from S would be combined with every value from the X column of R^+. In practice, however, only those tuples are combined where the Y value matches. The join of the intermediate result $R^+ \bowtie S$ with T can then be done in linear time in case a hash join is used, since T has no additional attributes, i.e. we just remove those tuples from $R^+ \bowtie S$ for which there is no join partner in T. For the other partition R^-, we apply a different join sequence. Due to our partitioning, every distinct X value in R^- occurs at most $\sqrt{n}$ times. Hence, we join R^- with T because there are at most $\sqrt{n}$ Y values in R^- for every tuple in T that matches on the X column. Again, this yields an intermediate result size bound of $O(n * \sqrt{n})$. As before, the final join – in this case with S – just eliminates tuples from the intermediate result $R^- \bowtie T$. Finally, we obtain the result of join query Q by taking the union of the two results $R^+ \bowtie S \bowtie T$ and $R^- \bowtie T \bowtie S$. If appropriate index structures are available prior to query execution, we can compute Q in time $O(n * \sqrt{n})$ [Got20].

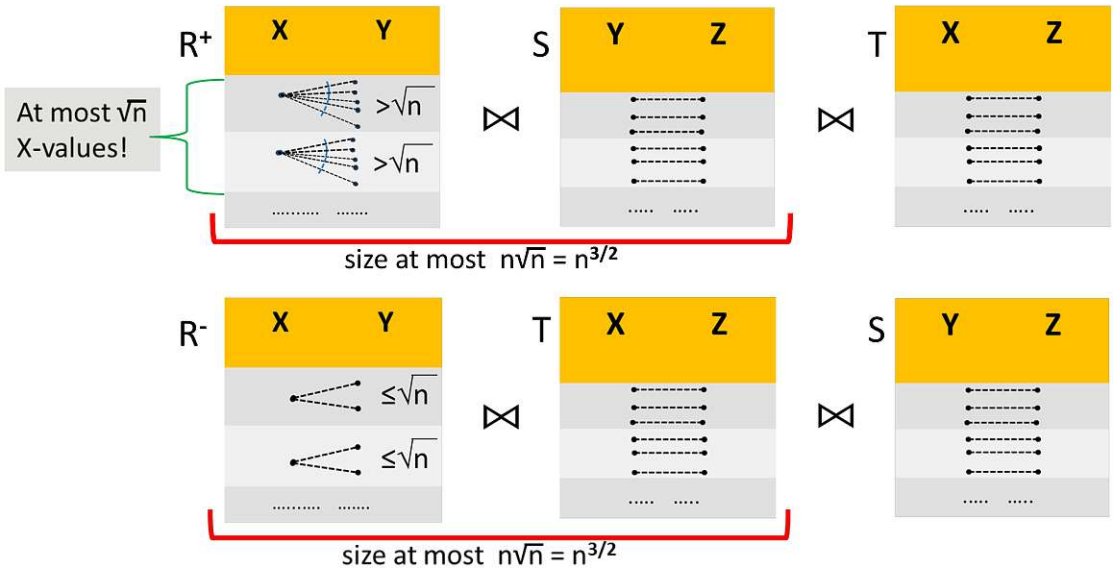


Figure 4.4: Partitioning-based algorithm applied to the join query $R(X,Y) \bowtie S(Y,Z) \bowtie T(X,Z)$ [Got20]

CHAPTER 5

Pandas in data science

In this chapter, we look at the Python ecosystem of tools for Data Science. In particular, we consider the currently most popular library for data preparation – Pandas. We explore possible reasons for its widespread use and study the design of Pandas. Finally, we identify other useful tools from different areas of data science that can be combined with Pandas.

According to a poll published by the website KDNuggets[1] in May 2018, Python has become the preferred programming language for the data science community, while R comes in second. Python's popularity most likely stems from (1) its ease of use and (2) the existence of a huge ecosystem of third-party libraries for every area of data science [SJ19].

Data preparation is an important step within a data science pipeline, as it aims to make the raw data usable for data analysis. This includes data cleansing, integration and transformation of raw data. Pandas is currently both the best and the most popular tool in this field because (1) it has an extensive documentation, (2) it has a wide range of functionality, and (3) it supports a variety of input/output data formats including Excel, CSV, SQL among others. Since other tools offer fewer features, for example, are limited to the HDF5 data format, their use is limited [SJ19].

According to McKinney [McK11], a major obstacle to widespread adoption of Python in the data science domain in the past has been the "lack of rich data structures with integrated handling of metadata". Metadata refers to the labelling information for data points in this context. For example, a table typically has labels for columns and possibly rows. Thus, in most cases, a row within a table can be uniquely identified by one or more labels. By using this metadata, various operations such as data manipulation, dataset integration, and aggregations can be specified in an intuitive way.

[1] https://www.kdnuggets.com

Pandas has closed this gap by providing its DataFrame abstraction for two-dimensional data. It is similar to R's `data.frame` class, which allows mixed-type data to be stored as a collection of independent columns while being flexible w.r.t. the size. The metadata for the two axes of a DataFrame object are stored in `Index` objects. In the simplest case, an index consists of a 1-dimensional vector of labels. While the column index consists of the column names, the row index could contain sequential numbers, although sorting is generally not required. Pandas stores both the indices and the contents of the columns of a DataFrame in n-dimensional NumPy[2] `ndarray` objects [McK11].

A so-called `BlockManager` is used, which serves three purposes. First, it facilitates the storage of columns with heterogeneous data types. It does this by managing multiple blocks, with each block containing a collection of columns with the same data type. Second, by this design, it can speed up row-oriented operations that involve only a single block, since all data is stored in a single `ndarray` object. Newly added columns are stored in new Block objects to avoid reallocate/copy steps when inserting/deleting columns. These are eventually consolidated into existing blocks of the same data type, either automatically when some operations are invoked on the DataFrame object, or when explicitly requested by the user. Third, by using the `BlockManager`, the user-facing API is separated from the implementation. This gives developers more freedom to modify the internal structures to achieve better performance or memory usage [McK11].

To speed up performance-sensitive operations, Pandas leverages the Cython language, which is built on top of Python and allows the invocation of C functions and the use of C variables and classes. Since C is a compiled language with lower-level memory access, critical parts of the code can be made many times faster. In addition, NumPy provides efficient vectorized computation, broadcasts over n-dimensional arrays, and fast implementations for many scientific algorithms written in C and Fortran [SJ19, McK11].

Pandas can be used in combination with tools used in other areas of data science thus enabling the creation of pipelines. Plotly [3], Matplotlib[4], and seaborn[5] are popular choices for creating plots in the data visualization domain, offering different levels of customization and ease of use. In machine learning, scikit-learn[6] is a widely used Python library. TensorFlow[7], PyTorch[8], and Keras[9] are popular in the machine learning field, the latter building on TensorFlow to increase accessibility with the drawback of limited customizability. The most popular tools in Big Data are Apache Spark[10] and Hadoop[11] MapReduce. These tools can be used to work with data stored on a cluster of machines.

[2] `https://numpy.org`
[3] `https://plotly.com/python/`
[4] `https://matplotlib.org`
[5] `https://seaborn.pydata.org`
[6] `https://scikit-learn.org/`
[7] `https://www.tensorflow.org`
[8] `https://pytorch.org`
[9] `https://keras.io`
[10] `https://spark.apache.org`
[11] `https://hadoop.apache.org`

The main difference between Spark and MapReduce is that Spark can work within memory, while MapReduce must always write to the file system [SJ19].

While we only consider Pandas for the development of a software prototype of a join optimizer in this thesis, the join ordering problem also exists for other data preparation tools such as R's `data.frame`.

CHAPTER 6

Related work

As far as the theoretical background of this work is concerned, the papers dealing with decomposition-based query answering and worst-case optimal joins are most relevant for our purposes. According to Gottlob et al. [GLL+23], decomposition-based query evaluation has been studied in many works [BDG07, CK19, CK21, CTG+21, CZB+22, GKSS22, LP22, GGGS07, SGL04] over the years. While we presented only two worst-case optimal join algorithms in Chapter 4, one of which we chose for our software prototype, there are other algorithms as well. The NPRR algorithm [NPRR12] by Ngo et al. was published before the Leapfrog Triejoin algorithm. In [NRR14], Ngo et al. later showed that these two algorithms are special cases of a more general algorithm that is also worst-case optimal. Navarro et al. recently proposed an algorithm that relies on quadtrees for representing the relations. By using this compact data structure, the algorithm can avoid the extra storage space required by other algorithms for indices such as B+ trees.

In the following, we consider three different systems in the Python ecosystem that we consider most comparable to the endeavors of this thesis. We show that these are either limited in portability due to their reliance on external dependencies or have additional overhead due to data transformations.

The Grizzly [HKS21] framework by Hagedorn et al. offers a Python API that is similar to the DataFrame of Pandas. It generates SQL statements, which are executed on an external database system. As a result, data needs to be already stored in the DBMS or is being transparently copied to it.

Aberger et al. proposed the EmptyHeaded [ALOR17, ALT+17] engine. A Jupyter interface allows a user to interact with the system by specifying DataFrames in Pandas as input and a SQL query to be executed over them. It supports conjunctive queries with selections and aggregations. EmptyHeaded transforms the query into a GHD (General Hypertree Decomposition), which is a generalization of a join tree beyond acyclicity. In addition, the Pandas DataFrames are transformed into a different data structure - namely

tries. Then, a C++ code based on the GHD is generated and executed. Finally, the result of the query is transformed back into a Pandas DataFrame. Aberger et al. improved upon this concept with the LevelHeaded [ALOR18] engine, enabling it to execute a larger set of queries. The authors mention that a WCOJ algorithm is used for computing the results of the individual nodes in the GHD. A cost-based optimizer is used for choosing the attribute-order as it has an impact on the performance. For computing the final join-result, the Yannakakis algorithm is used. Both EmptyHeaded and LevelHeaded are not first-class join implementations in Pandas, i.e. they do not work directly with the data structure of Pandas. Additionally, their solution is delivered as a Docker image[1] instead of a library, because their solution is built around a complex software architecture that involves many dependencies.

What makes this thesis different is that we aim to create a solution that works directly with the Pandas data structures and does not require SQL statements to be written. Additional overhead arising from data transformations should be reduced. Furthermore, another distinction is that we will not apply an optimization for cyclic join queries involving more than three relations. The solution should either be integrated directly into Pandas, or could be delivered as a Python package, e.g. via Python Package Index[2]. This increases the portability by not requiring an external dependency on a database system or a Docker image with the required software components. This would make it especially suitable for use in cloud notebooks, since the popular Google Colab offering does currently[3] not support the installation of Docker.

[1] https://hub.docker.com/r/craberger/emptyheaded/
[2] https://pypi.org
[3] https://github.com/googlecolab/colabtools/issues/299

CHAPTER 7

System architecture

In this chapter, we build on the theoretical foundations and discuss the system architecture of the developed software prototype of a join optimizer for Pandas. First, we consider the design of the `join_baseline` function which has been implemented for the evaluation, since Pandas does not provide a function that performs a natural join between a list of DataFrame objects if they do not all have the same column name(s). Then we will look at the very core of this work, the architecture of the function `join_optimised`, which applies some optimizations from database theory discussed in Chapter 3 and Chapter 4, with the goal of achieving better performance compared to the baseline. Finally, we outline some other implementation aspects, including an extension that allows repeated execution of queries with reuse of previous parameters, the requirements of our software prototype, and publishing considerations.

7.1 Baseline join function

A naive implementation would simply perform two-way joins in the order given by the user, with either an inner join being used if two relations share at least one common variable, or a cross join otherwise. However, this could lead to more cross joins being performed instead of inner joins, resulting in bigger intermediate result sizes, which negatively affects performance. To get a join function which can act as a more fair comparison during the evaluation later on, we compute the result with a more intelligent join order that minimizes cross joins with the following algorithm.

1. We create a graph $G = (V, E)$ which represents a given join query Q. The vertices $v \in V$ correspond to the atoms of Q. If there are multiple copies of an atom involved in the query, then there will also be multiple vertices. An edge $(u, v) \in E$ between the two vertices u and v indicates that the associated atoms share at least one variable, i.e. an inner join can be performed between them. The graph

could potentially have multiple connected components in case a cross-join is needed, but this is unlikely for most queries in practice. If there is only one connected component, then no cross-join is required.

2. Then, we utilize the graph to perform a DFS search for every connected component to derive an associated join sequence. As different join orders can have a significant difference in execution time, we introduce nondeterminism into this process so that repeated execution can give us different join orders. A random vertex is chosen as a starting point for the graph traversal in a connected component. During the DFS, we also select the next vertex to visit randomly from the set of unvisited neighbouring vertices of the current vertex.

3. Subsequently, we compute an intermediate result for each connected component by applying inner joins in the sequence determined in the prior step. For this, we utilize the `merge`[1] function of Pandas that enables us to execute a two-way inner join between two DataFrame objects. It performs a sort-merge join by default.

4. Finally, we use the `merge` function of Pandas to perform cross joins between the intermediate results associated with the connected components, yielding the final result.

7.2 Optimized join function

As part of this thesis, our focus is on enhancing the query execution time of acyclic join queries and the basic case of cyclic queries – triangle queries. We utilize the Yannakakis algorithm for evaluating acyclic queries efficiently. Furthermore, we have selected the partitioning-based algorithm for answering triangle queries – and other cyclic queries with three relations – due to (1) its simplicity, and (2) it not requiring specific index structures. This is an important difference to more sophisticated worst-case optimal join algorithms such as Leapfrog Triejoin. For cyclic queries with more than three relations we do not apply any optimizations. These are answered using the baseline join function. The overall algorithm of the optimized join function is illustrated in Figure 7.1. In the following, we will address implementation details for the different algorithms being used for the optimized join function.

GYO reduction & join tree construction

While the GYO reduction always yields the same result in the end, the atom elimination sequence and the selection of witnesses which are relevant for the join tree construction can be different. In order to obtain different join trees when a query is repeatedly executed, the list of remaining atoms that are considered for the reduction process is randomly shuffled before one of the two rules is (re)applied. Furthermore, with every

[1] `https://pandas.pydata.org/pandas-docs/version/1.5/reference/api/pandas.DataFrame.merge.html`

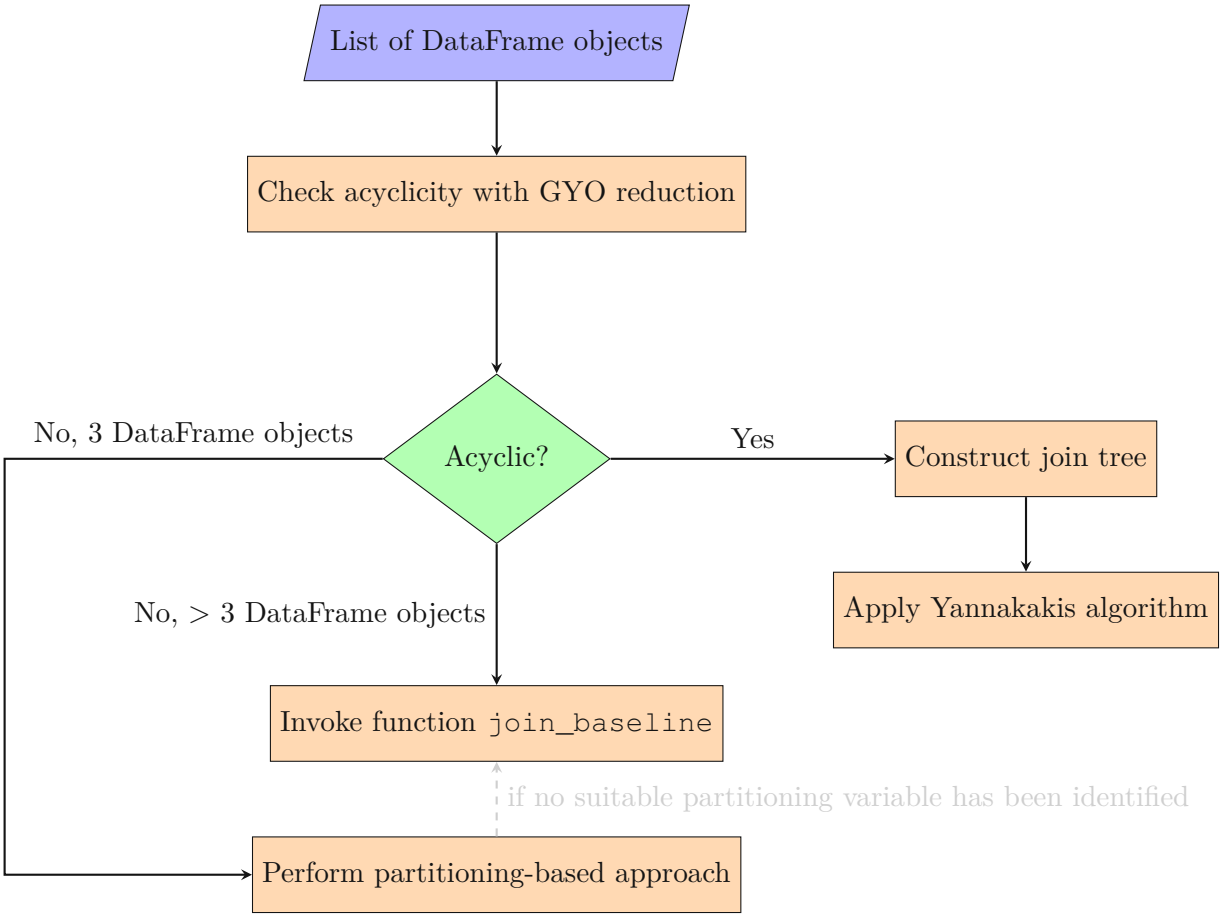


Figure 7.1: Algorithm of optimized join function `join_optimized`

step of the reduction, we always try to apply the rule of eliminating isolated hyperedges before eliminating redundant hyperedges.

If the GYO reduction terminates with an empty atom list, we know that the query is acyclic and use the elimination sequence and witness mapping to construct a join tree as follows. First, we consider the witness mapping and add an edge between an atom and its associated witness. Then, we add all missing atoms that occur in the elimination sequence to the graph. At this point, we potentially have a forest of trees. If this is the case, we merge the trees randomly by choosing some node of one connected component, connect it to some node of the next connected component and then in turn connect this node to some node of the subsequent one, and so on, until we have a single connected component. Finally, we root the tree at some node such that the resulting tree has minimal depth.

Yannakakis algorithm

The algorithm starts with a bottom-up traversal on the given join tree to compute left semi-joins between the relations associated to the parent and child nodes. For each inner node, we want to first perform the left semi-join where the right relation has the smallest number of tuples, since we expect this to produce the smallest intermediate result and thus the remaining semi-joins will be faster. In a regular post-order traversal [Val21] of a tree, the first child node of an inner node is visited first, followed by the second child node, and so on until finally the root is visited. However, to achieve our semi-join optimization, we need a variation of a post-order traversal where we first visit a child node of an inner node that has a path to a leave node with the highest depth in the subtree rooted at the current inner node. First, we perform a regular post-order DFS starting at the root node, determining an "inverse depth" for each node in the tree. For leaf nodes, the inverse depth is 1, while for inner nodes it is defined as the maximum of the inverse depth of the child nodes plus 1. Second, we perform an additional DFS starting at the root node, always visiting the child node with the highest inverse depth first. This results in a traversal sequence $[(u_1, v_1), (u_2, v_2), ..., (u_n, v_n)]$, where v_i is the parent of u_i for $1 \leq i \leq n$ with n corresponding to the number of nodes in the tree minus one since the root node does not have a parent. Third, we repeatedly remove a non-empty consecutive sublist of tuples from the beginning of the sequence where the parents are the same. For this sublist, we perform a reordering so that the tuple comes first that has an associated child relation with the smallest number of tuples. Finally, the semi-joins are performed for the sublist in the newly determined order. The process of removing sublists continues until we have performed the left semi-joins at the root node in the end.

While the `merge` function of Pandas has currently no dedicated support for the execution of a semi-join, there is ongoing work in a GitHub pull request[2] to integrate this into a future version. We have extracted the relevant code from the pull request and have added it to our implementation such that we can perform semi-joins in the meantime. However, this approach comes with the limitation that data type validations on join columns that are normally performed by Pandas as part of the `merge` function will not be executed. An alternative approach would have been to create a custom Pandas build that included the code for semi-joins in the `merge` function. However, this would have resulted in a more complex installation process and would prevent users from upgrading to newer Pandas versions.

If the DataFrame associated with the root node is empty after performing the semi-joins during the bottom-up traversal, we return an empty DataFrame object. Otherwise, we continue with a pre-order traversal [Val21] where we perform right semi-joins between the relations associated to the parent and child nodes. This means that the root node is visited first, followed by a traversal of the subtrees rooted at each of the child nodes from left to right.

[2] `https://github.com/pandas-dev/pandas/pull/49661`

Finally, we perform a second post-order traversal in the same order as before to compute the final result by performing inner or cross joins with the `merge` function of Pandas.

Partitioning-based algorithm

In our evaluation of cyclic queries with three relations, we choose the smallest relation R out of the three given relations for partitioning such that there exists a variable in R that exists also in one other relation T, but not also in S. In case no such variable exists, we answer the join query using the `join_baseline` function. If there are multiple candidate variables, then we pick the one with the smallest number of unique values in R by default, optionally the `join_optimized` function also enables random variable selection. We then split relation R into the heavy-hitters R^+ and light-hitters R^- partitions using the $\sqrt{n}$ threshold, where $|R| = n$.

Subsequently, we join R^+ first with the relation in which the partitioning variable does not occur, i.e. S. On the other hand, R^- is joined first with T, i.e. the relation which the partitioning variable appears in.

In case we have a triangle query of the form $R(X, Y) \bowtie S(Y, Z) \bowtie T(X, Z)$, both intermediate results would have already all variables that occur in the final result. Since we are not interested in duplicate results for this type of query, it would therefore be sufficient to perform a semi-join between the intermediate results and relations T respectively S. However, this behaviour might not be wanted by a user in general for a cyclic query involving three relations as this could cause a difference compared to performing a regular sequence of two-way joins. This is due to semi-joins not producing duplicates for every matching tuple in the other relation. Therefore, we perform an inner join by default in this case, and allow a user to optionally specify that a semi-join should be performed instead. If the intermediate results do not contain all the variables that appear in the final result, we must of course perform inner joins in any case.

Finally, we obtain the final result by taking the union of the two intermediate results.

7.3 Implementation

For evaluation purposes, we have added the possibility to repeat a query execution with the same strategy. This has been facilitated by adding a second return argument besides the DataFrame object that can be optionally fed as input to the `join_baseline` and `join_optimized` functions. A strategy hereby includes either a join order in case the baseline function has been applied; a join tree with a root node and associated pre-order and post-order traversal sequences in case the Yannakakis algorithm has been used; or the assignment of the given DataFrame objects to the R, S, T scheme and the specification of the partitioning variable if the partitioning-based algorithm has been utilized.

The Pandas join optimizer has been developed for Python 3.9 and higher. Beyond that, two Python packages must be installed on the system to ensure the successful execution

of our implementation. First, we recommend installing Pandas[3] version 1.5.3. Second, we use NetworkX[4] version 3.0, which is a Python package for working with graphs. It provides the necessary data structures for representing graphs and implements a variety of graph algorithms, including some relevant to our implementation.

Since the implementation does not rely solely on the Pandas package, we have decided that instead of integrating it directly into the Pandas library, the better approach is to enable our join optimizer to be published as a separate Python package. Therefore, we have already prepared the setup.py file required for this. The package can be uploaded to the Python Package Index (PyPi[5]), which provides a directory of Python software packages. After an upload to PyPi, users can then simply install the package with the `pip`[6] package installer and use one of the two join functions, provided that the other requirements for the Python environment are met. The source code for the implementation is available here: `https://github.com/steindlmedia/pd_join_optimizer`

[3] `https://pandas.pydata.org`
[4] `https://networkx.org`
[5] `https://pypi.org`
[6] `https://pip.pypa.io/`

CHAPTER 8

Empirical evaluation

In this chapter, we conduct the empirical evaluation of our software prototype. First, we describe which datasets and queries were selected for the evaluation. We then explain how we obtained the datasets and transformed respectively cleaned them so that they can be used for query execution. Next, we outline how the queries are executed, and specifically how the join functions described in the previous chapter are used. We then explain how the evaluation was performed. Finally, we present the results of our empirical evaluation.

8.1 Selection of datasets and queries

In order to acquire relevant data sets or queries, we looked at papers on related systems that claimed to improve the evaluation of acyclic queries or triangle queries.

Tu et al. have evaluated their query compiler DunceCap [TR15] with triangle queries on two datasets from the Stanford Network Analysis Project (SNAP) [LK14], Facebook[1] friend lists and Arxiv GR-QC[2] (General Relativity and Quantum Cosmology) collaboration network. Aberger et al. have performed the evaluation of their EmptyHeaded [ATOR16, ALT+17, ALOR17] prototype engine with triangle queries on the Google+[3] social circles, LiveJournal[4] social network, Orkut[5] social network and Patents [6] citation network datasets from SNAP. The statistics provided by SNAP for these selected datasets are given in Table 8.1.

[1] https://snap.stanford.edu/data/ego-Facebook.html
[2] https://snap.stanford.edu/data/ca-GrQc.html
[3] https://snap.stanford.edu/data/ego-Gplus.html
[4] https://snap.stanford.edu/data/soc-LiveJournal1.html
[5] https://snap.stanford.edu/data/com-Orkut.html
[6] https://snap.stanford.edu/data/cit-Patents.html

	Facebook	Arxiv	Google+	LiveJournal	Orkut	Patents
Nodes	4039	5242	107614	4847571	3072441	3774768
Edges	88234	14496	13673453	68993773	117185083	16518948
# Triangles	1612010	48260	1073677742	285730264	627584181	7515023

Table 8.1: Statistics provided by SNAP [LK14] for selected datasets containing triangles

We experimented with these datasets by loading them into a PostgreSQL database and executing the triangle query shown in Listing 8.1, which aims to find triangles in the dataset, with a triangle being defined as a set of three points connected by edges such that the first point is connected to the second, the second is connected to the third, and the third is connected to the first. We noticed that running the query on the Facebook dataset first gave an empty result. By adding the reversed edges to the dataset, we were able to get a non-empty result. We reason that only directed edges are stored in the dataset and adding the reversed edges results in an undirected set of edges which enables us to find triangles.

Listing 8.1: SQL for triangle query

```
SELECT a.x, a.y, b.y AS z
FROM dataset a
JOIN dataset b ON a.y = b.x
JOIN dataset c ON b.y = c.x
WHERE a.x = c.y;
```

To our surprise, we observed that the number of triangles found by the query for the Facebook and Arxiv datasets did not match the numbers given on the SNAP website for these datasets. We found multiples of the specified values, namely 9672060 respectively 289779 triangles. Due to the other SNAP datasets listed before being much larger, and the PostgreSQL database not having an optimization for this type of queries, we were not able to execute the triangle query on them, since we got a "No space left on device" error. Hence, we could not verify the number of triangles for these datasets and whether we need to add the reverse edges to find them. One possible explanation for the discrepancies could be that the statistics are outdated. Therefore, we will treat the results of the query execution with the PostgreSQL DB as ground truth for the correctness verification of the join execution with Pandas later on.

Aberger et al. have also evaluated their EmptyHeaded engine using a standard RDF workload, the LUBM [GPH05] benchmark. This benchmark provides a synthetic data generator that produces RDF data for a university ontology. The data generator has a scaling factor that is expressed in terms of the number of universities to generate. The authors of EmptyHeaded generated 133.6 million triples for their evaluation, which is equivalent to LUBM-1000[7]. Instead of storing the RDF data as triples in a single table, they used vertical partitioning as it turned out to deliver better performance. As

[7] https://www.w3.org/2007/OWL/wiki/images/c/cf/Zhe-f2f1.pdf

Dataset name	# Queries	# Acyclic queries	# Triangle queries
LUBM-1000	12	10	2
IMDB	14	14	0
Facebook	1	0	1
Arxiv	1	0	1
Google+	1	0	1
LiveJournal	1	0	1
Orkut	1	0	1
Patents	1	0	1

Table 8.2: Overview of collected datasets and queries

a result, only the triples that share a predicate name were stored in the same table. The benchmark includes mostly acyclic multiway star join patterns. There are only two cyclic queries that have a triangle pattern (LUBM queries 2 and 9). The evaluation was performed without the inference step for each query, as this is a standard practice for benchmarking comparisons, according to them. LUBM queries 6 and 10 were removed as they would correspond to the other queries in the benchmark without the inference step [ATOR16, ALT+17, ALOR17].

Fischl et al. [FGP16] have conducted an empirical study on hypergraph properties associated with CQs. For this purpose, they selected the conjunctive queries that are part of the JOB [LGM+15] query collection. The job benchmark uses an IMDB dataset.

Table 8.2 gives an overview of the data sets and the associated queries to be used for the empirical evaluation. It indicates how many of the queries are acyclic respectively have a triangle query pattern.

8.2 Dataset collection and cleansing

The Python script `get_datasets.py` has been created to enable a user to import all datasets chosen for the evaluation as CSV files into a specified directory, if they do not exist yet. This process involves (1) either downloading a dataset in compressed form from an external URL, or in the case of the LUBM benchmark, using a synthetic data generator, and (2) various transformations to ensure that the resulting CSV files can be imported into both Pandas DataFrame objects and a PostgreSQL DB without any data inconsistencies that may arise due to CSV parser behaviour differences. In the following we will outline the collection and cleansing pipeline for the different datasets.

To obtain a vertically-partitioned LUBM-1000 dataset in CSV format, we first use an improved[8] version of the associated data generator provided as a Java application for generating an ontology containing 1000 universities in the N-Triples format. After that

[8] `https://github.com/rvesse/lubm-uba`

we convert the N-Triples file to the CSV format using the `W3CNTriplesParser` of the `rdflib`[9] library. As part of this procedure, we remove the base URI of the ontology from subject, predicate and object. Beyond that, we perform vertical partitioning such that multiple CSV files are created, one for each predicate. Before we can invoke the parser, we need to remove all lines starting with <> from the N-Triples file using the application `sed`. This step ensures that the parser will not crash due to invalid lines occurring in the file. Finally, we remove duplicate lines from one of the generated CSV files, i.e. `type.csv`.

We download the IMDB dataset[10] provided by Leis et al [LGM+15]. Some of the CSV files are not compatible with the `read_csv`[11] function of Pandas. There are problems regarding the quotation and escape characters, e.g. if a `"` character appears in a column it should be escaped by another `"` character according to RFC-4180[12]. However, in file `company_name.csv` there are lines containing `\"` instead of `""`, for example. This causes an error when trying to import the CSV file with Pandas. We have tried adding the `quotechar='"'` and `escapechar='\\'` arguments to the `read_csv` function call. But, in this case the backslashes from column value `TBWA\CHIAT\DAY` were removed, which is not desired, as this led to a discrepancy in the results when executing a query using both Pandas and PostgreSQL. Replacing `\"` with `""` in the CSV files using the `sed` application also did not fix all issues. As a result, we decided to use a PostgreSQL Docker container to import the CSV files into a database and then re-export them as CSV files. In this way, we transform the CSV files so that they are interpreted in the same way by the parsers of Pandas and PostgreSQL. We thus ensure that the dataset can later be imported into both Pandas DataFrame objects and a PostgreSQL DB without any differences in the data, for example, a missing backslash character in one of them.

For the remaining graph datasets, which are only used to execute the triangle query, no transformations were required apart from the extraction of the downloaded compressed archives. This is due to these datasets containing only two numeric columns.

8.3 Query execution setup

The queries were obtained in SQL format, except for the LUBM queries, which were defined in Datalog syntax. In order to run the queries with Pandas, we had to translate them, and create executors.

The translation process involved creating a separate Python class for each query, containing functions for pre-processing and post-processing. For pre-processing, we first need to load the required DataFrame objects, i.e. the relations occurring in the query, from the dataset CSV files. Then, the loaded DataFrame objects must be filtered to match

[9] https://github.com/RDFLib/rdflib

[10] http://homepages.cwi.nl/~boncz/job/imdb.tgz

[11] https://pandas.pydata.org/pandas-docs/version/1.5/reference/api/pandas.read_csv.html

[12] https://www.ietf.org/rfc/rfc4180.txt

the selections specified in the query. Next, all columns are dropped from the DataFrame objects that are neither needed for the join operation nor for the final result. After that, we rename the column names so that a natural join can be performed. The preprocess function then returns a list of DataFrame objects that can be fed into one of our two join functions as input. The postprocess function accepts a single DataFrame object as input, i.e., the result of a join function. It involves dropping columns that should not be part of the final result, as well as renaming columns or performing aggregations if specified in the original query. A single DataFrame object is returned that represents the final result of the query execution.

Only for the queries belonging to the IMDB dataset, an aggregation has been defined in terms of the computation of the minimum value for each of the columns occurring in the result, which means that the result consists of only one row. However, we found that the minimum and maximum functions of Pandas behave differently compared to their counterpart in PostgreSQL, because Pandas apparently applies a different ordering of strings. Since this can lead to discrepancies when comparing the execution results, we have not performed any aggregation at all for the query executions of our evaluation.

We implemented three executors that can be used to evaluate a given query. Two of them rely on Pandas, i.e. we make use of the pre- and postprocessing functions of a query. Depending on the executor, either the `join_baseline` or `join_optimized` function is used for evaluating the join query.

Correctness verification

A third executor has been added to verify the correctness of the results obtained by the other two executors. This executor utilizes the Python package `sqlalchemy`[13] to connect to a PostgreSQL DB and to execute the SQL statement associated with a query. Upon use, the executor checks if a PostgreSQL Docker container is already running. If not, a new instance is created and an import of all datasets is performed. Alternatively, it is also possible to start a PostgreSQL Docker container instance before use with the interactive Python script `postgres_container.py`. It enables a user to import either a single dataset or all of them.

The correctness verification between two executors can be performed with the Python script `compare_executors.py`. The script executes all queries on both executors and compares the execution results. The comparison is done in such a way that the order of the rows and columns does not matter. If the DataFrame objects are not equal, the reason for the mismatch is determined as follows. First, a cumulative count of duplicates is added to both DataFrames so that we can ensure that we have the correct number of duplicates in each of them. If there is a tuple that occurs n times in a DataFrame, we add a sequential numbering so that the first tuple is assigned a 1 and the last n. Then, an outer join is computed between the two DataFrames using the `merge` function of

[13] `https://www.sqlalchemy.org`

Pandas. Finally, those rows are selected that are missing in one of the two DataFrames, and an exception is thrown with this information.

8.4 Evaluation setup

As part of our evaluation, we measure both the runtime and peak memory usage of the join operation during query execution. Since profilers incur additional overhead, we repeat each query execution a second time with a profiler attached to measure the peak memory usage.

We have chosen the Scalene [BSP22] profiler, which has recently been proposed by Berger et al. for the Python ecosystem and has already seen widespread adoption. With its novel sampling algorithm, it is able to simultaneously profile CPU, GPU, and memory usage while only having an overhead of up to 30%. Berger et al. claim that Scalene has the lowest overhead among accurate memory profilers. It provides information about memory usage over time, which is important to us because we have higher memory usage before the DataFrame filtering operations. This would mask the peak memory usage during the join operation if only one value would be reported by the profiler. We extract the peak memory usage of the join operation from the JSON report generated by Scalene as follows. We only consider the line-level information provided for the source code of the Python file containing the implementation of the two join functions. From this, we then calculate the maximum value for the `n_peak_mb` field.

The evaluation is started with the Python script `perform_evaluation.py`. A user can optionally specify (1) the number of runs to execute, (2) the strategy for selecting the partitioning variable, and (3) disabling duplicate reduction for the partitioning-based algorithm. In this context, duplicate reduction is not equivalent to the complete elimination of duplicates, as duplicates may still be produced by the first (inner) join that is performed for each of the two partitions. By default, each query is executed 10 times with both the baseline and the optimized executor. We use a different fixed random seed for each iteration to enable a more meaningful comparison of results among multiple evaluations with different parameters. We have disabled duplicate reduction during our evaluation because we found that the semi-join in its current implementation has a negative impact on the execution time of triangle queries. We have decided to run the Python script `execute_query.py` in a subprocess for each query execution for two reasons. First, this ensures that the evaluation process is not terminated by the operating system if a memory overflow occurs during a query execution. Second, this approach enables us to accurately capture the peak memory usage for each execution. The CLI script is given the query to execute, which executor to use for it, and the last two optional arguments listed before. If the subprocess is not started by Scalene, it (1) stores the measured runtime for executing the join function in a CSV file, and (2) stores the strategy used to execute the query in a pickle[14] file. The evaluation process then reads the CSV file and inserts the execution time into a DataFrame, which is later

[14] `https://docs.python.org/3/library/pickle.html`

exported as a CSV file. In case the CLI script is started with the profiler attached, the strategy of the previous run is loaded from the pickle file and passed to the executor so that it can be taken into account when the join function is called.

8.5 Results

The evaluation has been performed sequentially on a computer having an Apple M1 Pro SoC with 32 GB unified memory. We provide the raw results in Appendix B. Based on these results, we have computed the mean and standard deviation per query and executor for both the execution time and peak memory usage, which can be seen in Table 8.3 respectively Table 8.4. Note that most of these queries are acyclic, only the ones marked with the △ symbol are triangle queries. In the following, we will first analyse the results for acyclic queries, followed by an investigation of triangle queries.

Figure 8.1a illustrates the mean and standard deviation of the execution time for acyclic queries per executor. We can see that the optimized executor performs better than the baseline executor for the vast majority of acyclic queries. In addition, the execution times reported for the optimized executor tend to have lower variability. These results are notable because (1) a good join order, as mentioned earlier, is generally difficult for a user to choose and (2) the basic join function already performs a basic optimization to perform cross joins at the end, if that is even necessary. A naive user may not have considered this before when manually specifying a sequence of two-way joins, potentially yielding worse results than our baseline. We found that the optimized executor performed worse on only four of the 24 acyclic queries, i.e., queries 4c, 10b, 10c, and 32a on the IMDB dataset. From our set of acyclic queries, these four queries are among those with lower execution time, with a maximum mean value of 2276 ms for the optimized executor. Although lower variability was observed for this executor in these cases, its execution time was up to 68% longer compared to the baseline. For queries with longer execution times, the optimized executor was always faster than the baseline executor. The highest speed-up was observed for query 13d on the IMDB dataset, with a 4.68× speed-up. Notably, queries 16b and 17e on the IMDB dataset could only be executed with the optimized executor, as execution with the baseline executor resulted in an overflow of available memory on our test machine. As can be seen in Figure 8.1b, using the optimized executor resulted in a reduction of peak memory usage for the vast majority of queries and lower variability in general. Among all 24 queries, only queries 4c, 10b, and 32a on the IMDB dataset and query Q7 on the LUBM dataset exhibited higher peak memory usage with the optimized executor, the latter query showing a 49% increase. The most significant reduction in peak memory usage was seen for query 13d on the IMDB dataset, i.e., a reduction of 83% compared to the baseline value.

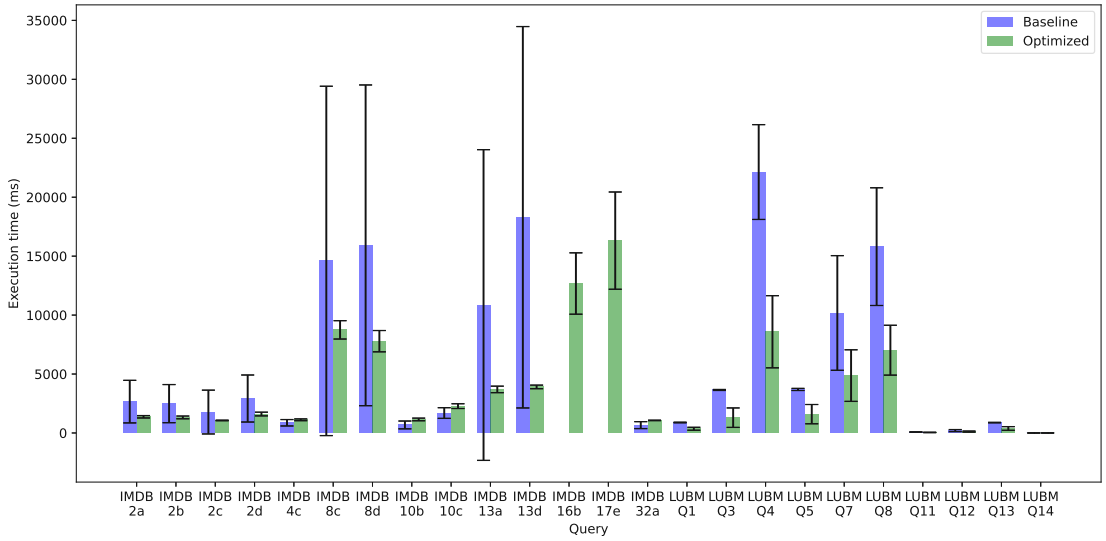
We now examine the results for the triangle queries. Figure 8.2a and Figure 8.2b visualize the mean and standard deviation of the execution time respectively the peak memory usage per executor for these queries. Whereas query execution on the Facebook dataset was accelerated by 13 % compared to the baseline, our optimized executor performed

Table 8.3: Execution times (mean ± standard deviation) in milliseconds for baseline and optimized executors

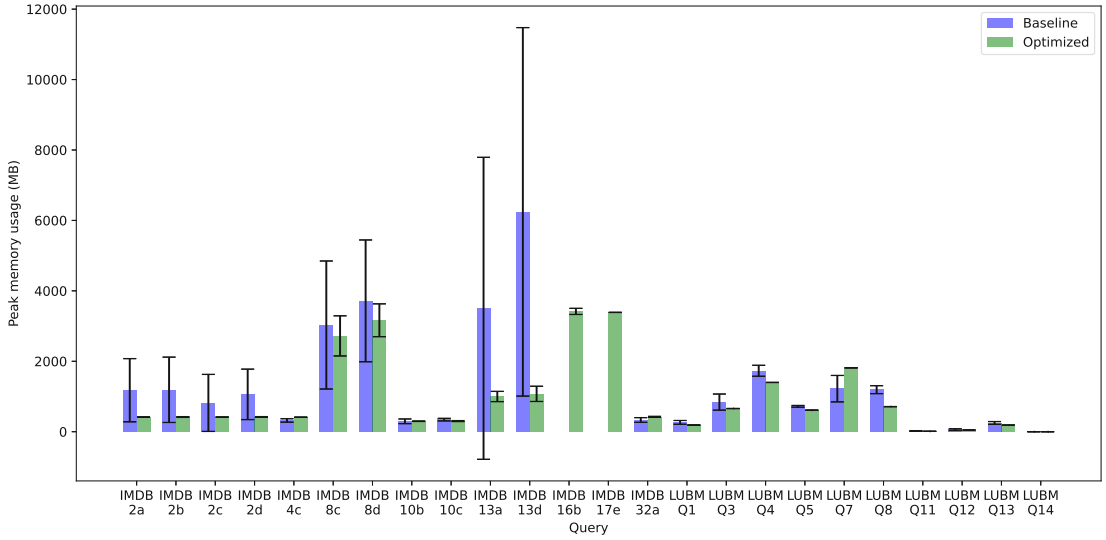
Dataset	Query	Baseline (ms)	Optimized (ms)
Arxiv	△	71.0± 3.1	78.1± 2.4
Facebook	△	2692.5±106.6	2347.5±107.9
IMDB	2a	2658.8±1805.5	1374.7±101.6
IMDB	2b	2489.4±1617.0	1319.3±115.5
IMDB	2c	1774.7±1857.5	1062.7± 23.5
IMDB	2d	2923.0±1994.7	1606.6±156.6
IMDB	4c	866.4±272.1	1118.4± 75.7
IMDB	8c	14 594.6±14 896	8747.3±779.0
IMDB	8d	15 918.2±13 603	7786.9±903.5
IMDB	10b	681.4±331.3	1147.2±111.3
IMDB	10c	1691.7±452.8	2276.4±202.6
IMDB	13a	10 853.7±13 189	3695.0±275.5
IMDB	13d	18 298.9±16 176	3906.9±151.0
IMDB	16b		12 677.7±2603
IMDB	17e		16 316.2±4125
IMDB	32a	665.9±294.4	1065.3± 19.4
LUBM	Q1	888.5± 24.4	357.1±123.6
LUBM	Q2△	17 096.5±14 426	17 782.3±11 467
LUBM	Q3	3647.6± 42.5	1298.2±825.5
LUBM	Q4	22 133.8±4021.8	8586.9±3060.7
LUBM	Q5	3691.6± 89.7	1597.5±817.0
LUBM	Q7	10 179.8±4863.9	4868.0±2188.7
LUBM	Q8	15 806.7±4993.0	7021.1±2114.4
LUBM	Q11	81.1± 3.2	45.3± 18.5
LUBM	Q12	202.1± 86.9	117.0± 44.8
LUBM	Q13	880.8± 19.3	382.0±155.6
LUBM	Q14	0.2	0.4

Table 8.4: Peak memory usage (mean ± standard deviation) in MB for baseline and optimized executors

Dataset	Query	Baseline (MB)	Optimized (MB)
Arxiv	△	44.8± 1.3	34.6± 4.9
Facebook	△	1055.7± 0.5	914.5± 3.5
IMDB	2a	1178.5±897.2	416.4± 3.4
IMDB	2b	1190.3±928.0	417.8± 10.7
IMDB	2c	817.9±809.6	415.7± 2.8
IMDB	2d	1060.8±716.0	418.5± 10.5
IMDB	4c	319.9± 49.0	410.9± 2.4
IMDB	8c	3030.4±1817.8	2721.3±570.0
IMDB	8d	3715.9±1729.4	3165.4±465.9
IMDB	10b	295.4± 66.0	297.0± 6.1
IMDB	10c	343.1± 36.2	302.2± 13.5
IMDB	13a	3503.8±4283.8	1000.0±145.1
IMDB	13d	6243.6±5232.0	1074.8±216.0
IMDB	16b		3416.4± 87.0
IMDB	17e		3387.8± 1.0
IMDB	32a	333.0± 66.9	421.6± 13.8
LUBM	Q1	265.2± 53.6	188.6± 6.5
LUBM	Q2△	1973.6±1844.7	2653.0±2365.2
LUBM	Q3	841.0±229.1	658.8
LUBM	Q4	1730.4±155.4	1398.9± 0.9
LUBM	Q5	720.7± 24.9	611.2
LUBM	Q7	1222.4±375.3	1815.4± 4.3
LUBM	Q8	1194.6±112.8	713.3± 0.1
LUBM	Q11	20.7± 9.8	11.6
LUBM	Q12	60.2± 24.3	48.8
LUBM	Q13	249.0± 39.4	187.9
LUBM	Q14	0.0	0.0



(a) Execution time

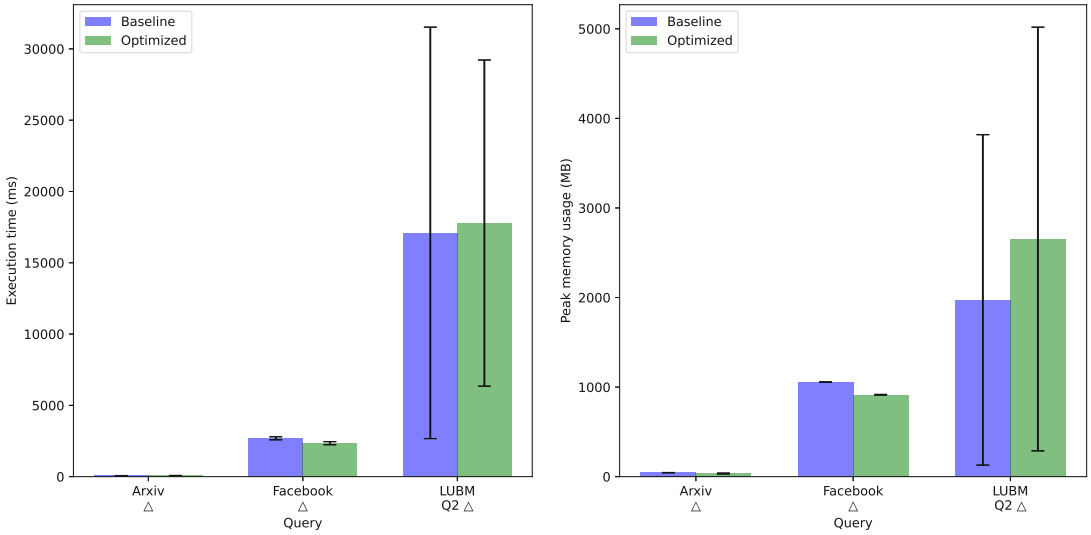


(b) Peak memory usage

Figure 8.1: Visualization of the mean and standard deviation of the execution time and the maximum memory consumption per executor for acyclic queries

worse on the other two triangle queries with an overhead ranging from 4 % to 10 %, although variability was around 20 % lower in both cases. In addition, we found the average peak memory usage to be 13 % and 23 % lower for the queries on the Facebook and Arxiv datasets, respectively. However, for the LUBM Q2 query, the memory consumption for the optimized executor is 34 % higher along with 28 % more variability. Unfortunately, we were unable to execute query Q9 on the LUBM dataset as well as the triangle query

on the Google+, LiveJournal, Orkut, and Patents datasets due to an overflow of available memory on our test system.



(a) Execution time

(b) Peak memory usage

Figure 8.2: Visualization of the mean and standard deviation of the execution time and the maximum memory consumption per executor for triangle queries

CHAPTER 9

Conclusion

9.1 Summary

In this thesis, we have investigated whether structure-based approaches from database theory, which have been shown to exhibit better runtime behaviour in theory, can also be used effectively in practice and, in particular, in the context of Pandas. To this end, we have extended Pandas with an open-source[1] implementation of a function that allows the user to specify a list of DataFrame objects to be joined with a natural join. Depending on the type of join query, we apply a different algorithm from database research with the intent to improve the execution time. For acyclic join queries, we use the Yannakakis algorithm. In case of cyclic join queries with three relations, we apply a partitioning-based worst-case optimal join algorithm with the goal of improving the execution time for triangle queries – and cyclic queries with three relations in general. Cyclic queries with more than three relations are beyond the scope of this work, so we do not apply optimizations for them.

We have conducted an empirical evaluation to validate the benefits of integrating the selected algorithms from database theory into Pandas. For this purpose, we have selected both synthetic and real-world datasets, including LUBM, IMDB, as well as some graph datasets. In total, we have executed 24 acyclic queries as well as triangle queries. Our experiments have shown that our optimized join function significantly improves execution time for most acyclic join queries through the use of the Yannakakis algorithm, while adding overhead to only a very small number of queries that have a lower execution time. The highest speed-up we have observed was 4.68 × compared to a baseline execution involving a sequence of two-way joins. Furthermore, we have seen lower peak memory usage for our optimized join function.

[1] https://github.com/steindlmedia/pd_join_optimizer

However, the worst-case optimal join algorithm we have chosen for triangle queries, which relies on partitioning a relation and applying different join sequences on the two partitions, has only been able to improve the execution time for one out of three triangle queries we have been able to run on our test machine. For the other two triangle queries, the use of this algorithm has resulted in an increase in execution time of up to 10%, although the variability has been about 20% lower compared to the baseline. We assume that this might be due to the datasets used in these queries not having enough skew to make a worst-case optimal join algorithm useful.

9.2 Future work

Worst-case optimal join algorithms are known to be particularly beneficial in situations with big skew. The fact that our worst-case optimal join implementation does not lead to a performance improvement in most cases is likely due to the absence of skew in the data. To improve the execution time for triangle queries, a promising approach would be to choose a different algorithm depending on the amount of skew.

In addition, the function we currently use to perform a semi-join (1) is not directly integrated into the merge function of Pandas, so the usual data type validations are not currently performed, and (2) has not been tuned for performance as it represents an initial PoC for integration. We assume that a future version of the semi-join function could also improve the execution time for the few acyclic queries on which our optimized join implementation is currently outperformed by the baseline.

In addition, we do not currently consider nearly acyclic queries. Future work could exploit hypertree decompositions in conjunction with the Yannakakis algorithm to improve the execution time for this broader class of queries.

Finally, we think that the concepts of decomposition-based query evaluation and worst-case optimal joins might also prove useful for other data preparation tools used in data science, e.g., R's `data.frame`.

APPENDIX A

Queries

Table A.1: Queries on IMDB dataset (based on [LGM+15])

Name	SQL syntax
2a	SELECT t.title AS movie_title FROM company_name AS cn, keyword AS k, movie_companies AS mc, movie_keyword AS mk, title AS t WHERE cn.country_code = '[de]' AND k.keyword = 'character-name-in-title' AND cn.id = mc.company_id AND mc.movie_id = t.id AND t.id = mk.movie_id AND mk.keyword_id = k.id AND mc.movie_id = mk.movie_id;
2b	SELECT t.title AS movie_title FROM company_name AS cn, keyword AS k, movie_companies AS mc, movie_keyword AS mk, title AS t WHERE cn.country_code = '[nl]' AND k.keyword = 'character-name-in-title' AND cn.id = mc.company_id AND mc.movie_id = t.id AND t.id = mk.movie_id AND mk.keyword_id = k.id AND mc.movie_id = mk.movie_id;
2c	SELECT t.title AS movie_title FROM company_name AS cn, keyword AS k, movie_companies AS mc, movie_keyword AS mk, title AS t WHERE cn.country_code = '[sm]' AND k.keyword = 'character-name-in-title' AND cn.id = mc.company_id AND mc.movie_id = t.id AND t.id = mk.movie_id AND mk.keyword_id = k.id AND mc.movie_id = mk.movie_id;
2d	SELECT t.title AS movie_title FROM company_name AS cn, keyword AS k, movie_companies AS mc, movie_keyword AS mk, title AS t WHERE cn.country_code = '[us]' AND k.keyword = 'character-name-in-title' AND cn.id = mc.company_id AND mc.movie_id = t.id AND t.id = mk.movie_id AND mk.keyword_id = k.id AND mc.movie_id = mk.movie_id;

Name	SQL syntax
4c	SELECT mi_idx.info AS rating, t.title AS movie_title FROM info_type AS it, keyword AS k, movie_info_idx AS mi_idx, movie_keyword AS mk, title AS t WHERE it.info = 'rating' AND k.keyword like '%sequel%' AND mi_idx.info > '2.0' AND t.production_year > 1990 AND t.id = mi_idx.movie_id AND t.id = mk.movie_id AND mk.movie_id = mi_idx.movie_id AND k.id = mk.keyword_id AND it.id = mi_idx.info_type_id;
8c	SELECT a1.name AS writer_pseudo_name, t.title AS movie_title FROM aka_name AS a1, cast_info AS ci, company_name AS cn, movie_companies AS mc, name AS n1, role_type AS rt, title AS t WHERE cn.country_code = '[us]' AND rt.role = 'writer' AND a1.person_id = n1.id AND n1.id = ci.person_id AND ci.movie_id = t.id AND t.id = mc.movie_id AND mc.company_id = cn.id AND ci.role_id = rt.id AND a1.person_id = ci.person_id AND ci.movie_id = mc.movie_id;
8d	SELECT an1.name AS costume_designer_pseudo, t.title AS movie_with_costumes FROM aka_name AS an1, cast_info AS ci, company_name AS cn, movie_companies AS mc, name AS n1, role_type AS rt, title AS t WHERE cn.country_code = '[us]' rt.role = 'costume designer' AND an1.person_id = n1.id AND n1.id = ci.person_id AND t.id = mc.movie_id AND mc.company_id = cn.id AND an1.person_id = ci.person_id AND ci.movie_id = mc.movie_id AND ci.movie_id = t.id AND ci.role_id = rt.id;
10b	SELECT chn.name AS character, t.title AS russian_mov_with_actor_producer FROM char_name AS chn, cast_info AS ci, company_name AS cn, company_type AS ct, movie_companies AS mc, role_type AS rt, title AS t WHERE ci.note like '%(producer)%' AND cn.country_code = '[ru]' AND rt.role = 'actor' AND t.production_year > 2010 AND t.id = ci.movie_id AND ci.movie_id = mc.movie_id AND chn.id = ci.person_role_id AND rt.id = ci.role_id AND cn.id = mc.company_id AND ct.id = mc.company_type_id AND t.id = mc.movie_id;
10c	SELECT chn.name AS character, t.title AS movie_with_american_producer FROM char_name AS chn, cast_info AS ci, company_name AS cn, company_type AS ct, movie_companies AS mc, role_type AS rt, title AS t WHERE ci.note like '%(producer)%' AND cn.country_code = '[us]' AND t.production_year > 1990 AND t.id = mc.movie_id AND ci.movie_id = mc.movie_id AND chn.id = ci.person_role_id AND rt.id = ci.role_id AND cn.id = mc.company_id AND ct.id = mc.company_type_id AND t.id = ci.movie_id;

Name	SQL syntax
13a	SELECT mi.info AS release_date, miidx.info AS rating, t.title AS german_movie FROM company_name AS cn, company_type AS ct, info_type AS it, info_type AS it2, kind_type AS kt, movie_companies AS mc, movie_info AS mi, movie_info_idx AS miidx, title AS t WHERE cn.country_code = '[de]' AND ct.kind = 'production companies' AND it.info = 'rating' AND it2.info = 'release dates' AND kt.kind = 'movie' AND mi.movie_id = t.id AND it2.id = mi.info_type_id AND kt.id = t.kind_id AND mc.movie_id = t.id AND cn.id = mc.company_id AND ct.id = mc.company_type_id AND miidx.movie_id = t.id AND it.id = miidx.info_type_id AND mi.movie_id = miidx.movie_id AND mi.movie_id = mc.movie_id AND miidx.movie_id = mc.movie_id;
13d	SELECT cn.name AS producing_company, miidx.info AS rating, t.title AS movie FROM company_name AS cn, company_type AS ct, info_type AS it, info_type AS it2, kind_type AS kt, movie_companies AS mc, movie_info AS mi, movie_info_idx AS miidx, title AS t WHERE cn.country_code = '[us]' AND ct.kind = 'production companies' AND it.info = 'rating' AND it2.info = 'release dates' AND kt.kind = 'movie' AND mi.movie_id = t.id AND it2.id = mi.info_type_id AND kt.id = t.kind_id AND mc.movie_id = t.id AND cn.id = mc.company_id AND ct.id = mc.company_type_id AND miidx.movie_id = t.id AND it.id = miidx.info_type_id AND mi.movie_id = miidx.movie_id AND mi.movie_id = mc.movie_id AND miidx.movie_id = mc.movie_id;
16b	SELECT an.name AS cool_actor_pseudonym, t.title AS series_named_after_char FROM aka_name AS an, cast_info AS ci, company_name AS cn, keyword AS k, movie_companies AS mc, movie_keyword AS mk, name AS n, title AS t WHERE cn.country_code = '[us]' AND k.keyword = 'character-name-in-title' AND an.person_id = n.id AND n.id = ci.person_id AND ci.movie_id = t.id AND t.id = mk.movie_id AND mk.keyword_id = k.id AND t.id = mc.movie_id AND mc.company_id = cn.id AND an.person_id = ci.person_id AND ci.movie_id = mc.movie_id AND ci.movie_id = mk.movie_id AND mc.movie_id = mk.movie_id;
17e	SELECT n.name AS member_in_charnamed_movie FROM cast_info AS ci, company_name AS cn, keyword AS k, movie_companies AS mc, movie_keyword AS mk, name AS n, title AS t WHERE cn.country_code = '[us]' AND k.keyword = 'character-name-in-title' AND n.id = ci.person_id AND ci.movie_id = t.id AND t.id = mk.movie_id AND mk.keyword_id = k.id AND t.id = mc.movie_id AND mc.company_id = cn.id AND ci.movie_id = mc.movie_id AND ci.movie_id = mk.movie_id AND mc.movie_id = mk.movie_id;
32a	SELECT lt.link AS link_type, t1.title AS first_movie, t2.title AS second_movie FROM keyword AS k, link_type AS lt, movie_keyword AS mk, movie_link AS ml, title AS t1, title AS t2 WHERE k.keyword = '10,000-mile-club' AND mk.keyword_id = k.id AND t1.id = mk.movie_id AND ml.movie_id = t1.id AND ml.linked_movie_id = t2.id AND lt.id = ml.link_type_id AND mk.movie_id = t1.id;

Table A.2: Queries on LUBM dataset (based on [ALT+17])

Name	Datalog syntax
Q1	`out (x) ← type (x, 'GraduateStudent'), takesCourse (x, 'Dept0.Uni0.edu/GraduateCourse0').`
Q2	`out (x, y, z) ← memberOf (x, y), subOrganizationOf (y, z), undergraduateDegreeFrom (x, z), type (x, 'GraduateStudent'), type (y, 'Department'), type (z, 'University').`
Q3	`out (x) ← publicationAuthor (x, 'Dept0.Uni0.edu/AssistantProf0'), type (x, 'Publication').`
Q4	`out (x, y, z, w) ← worksFor (x, 'Dept0.Uni0.edu'), name (x, y), emailAddress (x, w), telephone (x, z), type (x, 'AssociateProf').`
Q5	`out (x) ← type (x, 'UndergraduateStudent'), memberOf (x, 'Dept0.Uni0.edu').`
Q7	`out (x, y) ← teacherOf ('Dept0.Uni0.edu/AssociateProf0', x), takesCourse (y, x), type (x, 'Course'), type (y, 'UndergraduateStudent').`
Q8	`out (x, y, z) ← memberOf (x, y), emailAddress (x, z), type (x, 'UndergraduateStudent'), type (y, 'Department') subOrganizationOf(y, 'Uni0.edu').`
Q9	`out (x, y, z) ← type (x, 'UndergraduateStudent'), type (y, 'Course'), type (z, 'AssistantProf'), advisor (x, z), teacherOf (z, y), takesCourse (x, y).`
Q11	`out (x) ← type (x, 'ResearchGroup'), subOrganizationOf(x, 'Uni0.edu').`
Q12	`out (x, y) ← worksFor (y, x), type (y, 'FullProf'), subOrganizationOf (x, 'Uni0'), type (x, 'Department').`
Q13	`out (x) ← type (x, 'GraduateStudent'), undergraduateDegreeFrom(x, 'University567.edu').`
Q14	`out (x) ← type (x, 'UndergraduateStudent').`

For some equality conditions, abbreviations have been used to improve readability.

APPENDIX B

Results

Table B.1: Query execution time and peak memory usage

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
Arxiv	△	baseline	76.6	44.4
			68.6	48.5
			68.6	44.4
			68.7	44.4
			74.7	44.4
			68.7	44.4
			70.6	44.4
			74.8	44.4
			70.0	44.4
			68.8	44.4
		optimized	78.9	43.8
			81.4	32.3
			81.2	43.8
			79.3	32.3
			79.0	32.3
			76.7	32.3
			75.4	32.3
			75.5	32.3
			78.5	32.3
			74.6	32.3
Facebook	△	baseline	2710.7	1055.5
			2596.0	1055.5
			2628.6	1055.5
			2634.8	1055.5
			2574.4	1055.5
			2582.5	1055.5
			2805.1	1055.5
			2704.1	1055.5
			2848.8	1055.5
			2840.6	1057.1
		optimized	2290.0	913.1
			2336.1	913.1
			2304.8	913.1
			2628.7	913.1
			2313.9	913.1
			2432.0	913.1

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			2296.1	916.2
			2293.2	913.1
			2284.6	924.1
			2295.7	913.1
IMDB	2a	baseline	3892.8	2105.9
			6057.4	2043.7
			819.6	310.6
			1039.5	292.5
			3891.8	2105.9
			1232.0	356.9
			1048.1	292.5
			3473.7	1756.5
			1257.0	414.7
			3876.2	2105.9
		optimized	1165.8	418.4
			1426.7	420.9
			1428.9	420.9
			1418.4	413.0
			1200.8	418.4
			1413.4	413.0
			1411.5	413.0
			1441.8	414.6
			1418.6	413.0
			1421.4	418.4
	2b	baseline	3863.1	2105.9
			783.1	271.7
			784.3	261.2
			3879.2	2068.3
			4036.4	2050.6
			4158.2	2073.0
			4142.4	2050.6
			1255.3	414.7
			977.5	324.4
			1014.9	282.6
		optimized	1169.2	418.4
			1376.8	413.1
			1386.8	417.0
			1388.2	416.9
			1376.7	413.0
			1421.7	413.0
			1376.1	447.6
			1144.4	413.0
			1146.8	413.0
			1405.8	413.0
	2c	baseline	5857.3	2041.9
			3879.7	2105.9
			786.4	310.6
			785.9	310.6
			784.0	347.1
			804.9	310.6
			3099.7	1810.6
			519.0	310.6
			784.9	310.6
			445.2	320.3
		optimized	1035.1	418.4
			1035.3	418.4
			1067.8	413.0
			1099.8	413.1
			1072.7	418.4
			1070.0	418.4
			1048.7	418.4

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			1065.7	413.0
			1036.8	413.0
			1094.9	413.0
	2d	baseline	2214.9	751.2
			2606.5	894.9
			2593.7	894.9
			810.8	424.8
			806.9	302.2
			5839.8	2045.0
			810.9	347.1
			2588.0	846.2
			5539.3	2050.6
			5419.6	2050.6
		optimized	1489.5	413.0
			1730.2	418.4
			1542.0	413.0
			1839.0	417.2
			1837.2	418.4
			1484.9	413.1
			1720.0	418.4
			1466.3	413.0
			1488.0	447.6
			1469.0	413.1
	4c	baseline	930.8	298.4
			1413.4	414.9
			940.7	297.7
			596.2	236.1
			1105.5	356.8
			938.7	337.3
			602.0	313.9
			944.1	318.5
			592.1	350.4
			600.4	275.0
		optimized	1172.1	409.2
			1163.6	406.2
			1166.2	409.6
			965.6	413.0
			1106.2	413.0
			1106.2	413.0
			1007.8	413.1
			1198.7	409.6
			1158.8	409.6
			1138.8	413.0
	8c	baseline	47860.4	6425.7
			5067.2	899.7
			5238.4	1833.1
			5735.2	1832.4
			35154.3	5785.5
			11824.9	3550.9
			12804.4	3007.5
			5284.3	1833.1
			5520.4	1833.1
			11456.4	3303.4
		optimized	9924.1	3379.7
			7777.1	2281.3
			8495.6	2281.3
			8191.7	2280.2
			8483.2	2281.4
			7843.2	2273.6
			9213.1	3387.5
			9117.9	3387.4

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			9950.8	3379.7
			8476.6	2281.4
	8d	baseline	7099.9	3098.9
			2757.8	1833.1
			2636.1	899.7
			9107.7	3028.0
			6856.2	3105.0
			31849.7	5763.1
			33156.3	5327.2
			34603.5	5776.5
			25785.5	5342.7
			5329.2	2985.4
		optimized	8329.7	3387.4
			8466.5	3379.7
			6225.5	2281.3
			8142.2	3387.5
			8479.7	3388.3
			7956.9	3387.4
			5994.6	2281.4
			8026.5	3387.5
			8226.5	3386.2
			8021.0	3387.5
	10b	baseline	645.6	291.2
			400.7	194.0
			535.0	217.9
			990.7	334.5
			872.4	289.8
			549.8	361.5
			1432.1	364.7
			326.7	227.5
			499.2	291.2
			561.5	382.2
		optimized	1162.3	294.9
			1207.3	298.1
			848.3	290.5
			1199.7	298.1
			1198.5	287.5
			1194.8	298.1
			1124.1	294.9
			1187.1	304.2
			1233.7	308.7
			1116.1	294.9
	10c	baseline	2386.2	373.1
			1877.4	365.2
			932.4	316.6
			1974.2	359.5
			1429.3	309.7
			1499.6	418.6
			1085.3	315.2
			2061.9	338.2
			1733.9	302.1
			1936.5	332.5
		optimized	2493.8	290.6
			2444.6	290.6
			2270.4	320.8
			2493.3	290.6
			2169.0	293.6
			2419.5	322.4
			2009.9	293.6
			2009.6	314.8
			2410.9	311.8

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			2043.1	293.5
	13a	baseline	43146.5	14620.4
			5573.7	1188.3
			3035.3	1072.7
			9023.1	2500.4
			2858.6	1160.5
			3666.4	1018.6
			2416.2	1545.9
			3968.0	1006.2
			10087.3	5146.6
			24761.3	5777.9
		optimized	3142.5	1396.8
			3699.7	1068.9
			3890.6	940.9
			3547.3	940.9
			3437.1	940.9
			3988.0	944.2
			3852.0	941.3
			4033.9	940.9
			3792.5	940.9
			3566.6	944.1
	13d	baseline	32232.7	13454.5
			3432.8	1102.4
			32252.8	13457.0
			15201.7	3937.4
			14736.3	5329.3
			53564.1	13262.6
			5223.6	1714.2
			8358.3	2682.5
			13259.0	6470.1
			4727.8	1025.5
		optimized	3651.8	944.2
			3779.8	941.6
			3911.6	940.9
			4114.2	934.8
			3832.4	940.9
			3851.7	1387.9
			3934.1	1389.4
			4173.7	940.9
			3893.5	940.9
			3926.1	1386.1
	16b	optimized	11341.7	3387.3
			10840.2	3387.4
			13467.5	3395.2
			13632.9	3387.5
			16997.7	3664.0
			14870.2	3390.5
			8369.4	3386.7
			14041.6	3390.5
			9593.5	3387.4
			13622.8	3387.5
	17e	optimized	16760.3	3390.5
			20938.7	3387.4
			16492.2	3387.6
			17681.1	3387.4
			21768.8	3387.4
			16267.6	3387.5
			10344.9	3387.4
			8527.0	3387.4
			18219.1	3387.4
			16162.0	3387.5

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
	32a	baseline	277.5	313.9
			586.9	325.2
			913.2	516.4
			889.8	323.7
			932.2	284.8
			921.8	325.9
			561.9	313.9
			309.0	325.1
			976.1	276.3
			290.4	325.1
		optimized	1031.5	416.1
			1067.5	417.2
			1095.6	447.6
			1065.8	416.1
			1087.8	413.1
			1060.5	416.1
			1051.7	413.0
			1063.0	415.8
			1047.1	413.1
			1082.2	447.6
LUBM	Q1	baseline	933.6	350.8
			868.2	230.3
			858.9	230.3
			896.8	246.2
			909.5	246.2
			852.1	230.3
			880.8	379.5
			892.0	246.2
			899.0	246.2
			894.4	246.2
		optimized	215.0	183.5
			457.2	196.2
			455.5	196.2
			446.8	183.6
			446.7	196.2
			455.5	196.2
			217.8	183.6
			455.0	183.6
			211.5	183.5
			209.7	183.6
	$Q2^{\triangle}$	baseline	10527.5	1070.6
			11302.7	1033.1
			10001.4	1033.1
			10329.5	1394.6
			44313.7	5467.8
			9606.6	1086.1
			11519.9	1033.1
			9057.3	1079.0
			44551.6	5467.8
			9755.0	1070.6
		optimized	9449.2	1107.2
			34426.4	6076.4
			12898.5	1389.2
			34476.8	6076.4
			12053.0	1389.2
			10567.8	1150.4
			8335.2	1107.2
			33757.6	6076.4
			9336.5	1079.0
			12521.7	1079.1
	Q3	baseline	3599.3	717.2

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			3585.4	717.2
			3681.6	717.2
			3683.8	732.2
			3679.9	1274.3
			3581.2	781.3
			3645.9	732.2
			3661.8	732.2
			3675.0	1274.3
			3682.2	732.2
		optimized	2265.9	658.8
			2276.3	658.9
			658.9	658.8
			2206.8	658.9
			661.1	658.8
			2279.2	658.9
			660.8	658.8
			652.8	658.8
			661.1	658.8
			659.3	658.8
	Q4	baseline	25788.1	1938.4
			19760.5	1627.2
			19870.9	1627.3
			19764.7	1718.5
			20041.6	1621.7
			19570.6	1625.5
			19768.0	1604.0
			19786.4	1627.2
			25953.0	1938.4
			31034.1	1975.5
		optimized	5546.8	1397.5
			15493.1	1397.7
			6812.4	1399.5
			7459.5	1399.5
			9187.5	1399.4
			7475.0	1399.5
			10124.3	1399.5
			5679.4	1399.5
			6796.0	1399.5
			11294.7	1397.6
	Q5	baseline	3787.0	716.9
			3601.3	702.0
			3806.2	716.9
			3607.0	702.0
			3803.1	716.9
			3640.3	766.0
			3628.5	766.0
			3626.0	702.0
			3782.2	716.9
			3634.7	702.0
		optimized	2238.9	611.2
			2226.9	611.2
			2214.7	611.2
			645.9	611.2
			651.2	611.2
			2222.1	611.2
			648.1	611.1
			647.9	611.2
			2235.9	611.2
			2243.5	611.2
	Q7	baseline	7303.9	1053.1
			7327.3	1053.1

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			17244.2	1753.3
			6742.1	849.1
			7372.3	1053.1
			7332.8	1053.1
			7355.2	1053.1
			6706.8	849.1
			17257.2	1753.3
			17156.1	1753.3
		optimized	3166.8	1810.5
			7436.6	1818.7
			3244.2	1818.7
			7349.0	1818.7
			7446.1	1818.7
			3012.2	1810.5
			3226.0	1810.4
			3205.9	1818.7
			3185.7	1810.4
			7407.8	1818.7
	Q8	baseline	19364.0	1242.8
			15481.0	1201.4
			9659.9	1035.1
			15036.7	1285.1
			9391.2	1035.4
			20274.0	1274.2
			15443.5	1277.7
			22385.4	1274.2
			9661.7	1035.2
			21369.5	1285.1
		optimized	4900.2	713.2
			7324.0	713.4
			7726.6	713.3
			10456.4	713.3
			4910.4	713.5
			4924.8	713.2
			10094.7	713.5
			7638.9	713.3
			7349.6	713.4
			4885.8	713.5
	Q11	baseline	79.2	10.4
			81.2	10.4
			83.4	24.0
			84.1	24.0
			81.5	34.7
			83.6	24.0
			78.3	10.4
			85.1	24.0
			80.4	35.0
			74.5	10.4
		optimized	30.0	11.6
			31.5	11.6
			70.4	11.6
			33.3	11.6
			66.7	11.6
			65.5	11.6
			30.3	11.6
			30.5	11.6
			30.5	11.6
			64.0	11.6
	Q12	baseline	122.1	36.6
			299.9	91.7
			298.0	91.2

Dataset	Query	Executor	Execution time (ms)	Peak memory usage (MB)
			267.5	92.6
			118.9	36.6
			120.5	36.6
			120.2	36.6
			122.5	54.4
			298.6	59.2
			252.9	66.3
		optimized	92.9	48.9
			97.5	48.9
			202.5	48.8
			94.7	48.8
			93.8	48.8
			103.3	48.8
			94.6	48.8
			92.5	48.9
			201.1	48.8
			97.1	48.8
	Q13	baseline	898.6	232.4
			869.1	250.2
			899.4	232.4
			862.5	219.3
			872.5	241.2
			860.7	241.2
			863.6	241.2
			868.7	241.2
			909.4	358.5
			903.0	232.4
		optimized	507.4	187.9
			497.8	187.9
			502.7	187.9
			203.0	187.9
			506.0	187.9
			503.8	187.9
			201.7	187.9
			497.1	187.9
			198.9	187.9
			201.5	187.9
	Q14	baseline	0.2	0.0
			0.2	0.0
			0.2	0.0
			0.2	0.0
			0.2	0.0
			0.2	0.0
			0.2	0.0
			0.2	0.0
			0.2	0.0
		optimized	0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0
			0.4	0.0

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAA+22] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, Anhai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca Popa, Raghu Ramakrishnan, Christopher Re, Michael Stonebraker, and Dan Suciu. The Seattle report on database research. *Communications of the ACM*, 65(8):72–79, July 2022.

[AGM13] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM Journal on Computing*, 42(4):1737–1767, January 2013. Publisher: Society for Industrial and Applied Mathematics.

[ALOR17] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Mind the gap: bridging multi-domain query workloads with EmptyHeaded. *Proceedings of the VLDB Endowment*, 10(12):1849–1852, August 2017.

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

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

[ATOR16] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. Old Techniques for New Join Algorithms: A Case Study in RDF Processing, February 2016. arXiv:1602.03557 [cs].

[BC81] Philip A. Bernstein and Dah-Ming W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, January 1981.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic*, pages 208–222, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

[BMT20] Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *The VLDB Journal*, 29(2-3):655–679, May 2020.

[BSP22] Emery D. Berger, Sam Stern, and Juan Altmayer Pizzorno. Triangulating python performance issues with scalene, 2022.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '98*, pages 34–43, Seattle, Washington, United States, 1998. ACM Press.

[CK19] Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, 64(5):828–860, August 2019.

[CK21] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. *ACM Transactions on Database Systems*, 46(2):1–41, May 2021.

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

[CTG+21] Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, June 2021.

[CZB+22] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. *ACM Transactions on Database Systems*, 47(3):1–49, August 2022.

[DIR07] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive Query Processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[EN15] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 7th edition, 2015.

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

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

[FGP16] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and Fractional Hypertree Decompositions: Hard and Easy Cases, November 2016.

[GGGS07] Lucantonio Ghionna, Luigi Granata, Gianluigi Greco, and Francesco Scarcello. Hypertree decompositions for query optimization. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, April 2007.

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

[GKSS22] Gaetano Geck, Jens Keppeler, Thomas Schwentick, and Christopher Spinrath. Rewriting with acyclic queries: Mind your head. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

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

[GLPR21] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Complexity analysis of generalized and fractional hypertree decompositions. *Journal of the ACM*, 68(5):1–50, September 2021.

[GLS01a] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, May 2001.

[GLS01b] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: A Survey. In Jiří Sgall, Aleš Pultr, and Petr Kolman, editors, *Mathematical Foundations of Computer Science 2001*, Lecture Notes in Computer Science, pages 37–57, Berlin, Heidelberg, 2001. Springer.

[GM06a] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, SODA '06, pages 289–298, USA, January 2006. Society for Industrial and Applied Mathematics.

[GM06b] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers – soda 2006. https://www.cs.bme.hu/ dmarx/papers/marx-soda2006-slides.pdf, January 2006.

[GMS09] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *Journal of the ACM*, 56(6):1–32, September 2009.

[Got20] Georg Gottlob. Cpaior 2020 invited talk: Georg gottlob. https://www.youtube.com/watch?v=rZdIFbGgBNI, September 2020.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, 2005. Selcted Papers from the International Semantic Web Conference, 2004.

[Gra80] MH Graham. *On the universal relation*. University of Toronto. Computer Systems Research Group, 1980.

[HKS21] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. Putting pandas in a box. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[LGM+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, November 2015.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[LP22] Carsten Lutz and Marcin Przybylko. Efficiently enumerating answers to ontology-mediated queries. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, June 2022.

[McK11] Wes McKinney. pandas: a foundational python library for data analysis and statistics. 2011.

[NPRR12] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, page 37–48, New York, NY, USA, 2012. Association for Computing Machinery.

[NRR14] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, February 2014.

[Pic21] Reinhard Pichler. Conjunctive queries. Slide set from lecture 181.140 Database Theory of University of Technology, Vienna, 2021.

[RS84] Neil Robertson and P.D Seymour. Graph minors. III. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, February 1984.

[RS86] Neil Robertson and P.D Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, September 1986.

[SGL04] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, June 2004.

[SJ19] I. Stančin and A. Jović. An overview and comparison of free python libraries for data mining and big data analysis. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 977–982, 2019.

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

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, August 1984.

[Val21] Gabriel Valiente. *Algorithms on Trees and Graphs: With Python Code*. Texts in Computer Science. Springer International Publishing, Cham, 2021.

[Vel14] Todd Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm, 2014. Type: dataset.

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

[YO79] C.T. Yu and M.Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's Third International Applications Conference, 1979.*, pages 306–312, November 1979.