TU WIEN Informatics

A Structural Approach to Query Optimisation for Efficient Join and Aggregate Processing

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Alexander Selzer
Registration Number 01633655

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Reinhard Pichler

The dissertation has been reviewed by:

Zoltan Miklos	Andrea Calí

Vienna, November 23, 2025

Alexander Selzer

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

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Alexander Selzer

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.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 23. November 2025

Alexander Selzer

Abstract

Join processing remains a fundamental challenge in database systems, particularly when intermediate results grow large. This problem is frequently encountered in analytical queries, which aggregate data over many relations, but output only a comparably small final result. Although the vast majority of user-generated queries are acyclic, and the foundational result by Yannakakis provides a solution for avoiding unnecessary computation of intermediate results in theory, structure-guided query evaluation remains absent from standard database systems. In this thesis, we work towards closing this gap from theory to practice.

To explore the benefits and limitations of structure-guided query processing, we introduce a query-rewriting-based implementation of Yannakakis' algorithm, on top of different DBMSs. Additionally, we identify the class of zero-materialisation answerable (0MA) queries, which can be answered efficiently without materialising join results. Experimental evaluation shows that the rewriting approach can achieve significant speedups on hard instances but fails to outperform in most situations. These results motivate us to frame the problem as an algorithm selection problem – we therefore apply machine learning to solve this problem, setting up a new data set of queries in the process, and verify that we can reliably decide when to apply the rewriting.

Following up on these encouraging results, we go beyond the quite restrictive 0MA class, introducing *guarded* and *piecewise-guarded* aggregate queries, allowing us to cover a wide range of aggregate queries while avoiding materialisation. We integrate these optimisations into the query optimiser of Spark SQL by implementing logical optimisations and a new physical operator *AggJoin*. Through experimental evaluation, we show that this implementation speeds up aggregate processing on many queries of several benchmarks while never degrading performance. Furthermore, to cover arbitrary unguarded queries, where materialisation cannot be fully avoided, we extend the *AggJoin* to the *GroupAggJoin* operator, substantially reducing materialisation on these queries as well. By establishing a new benchmark for unguarded queries, we show that this approach successfully handles moderately unguarded queries.

Having established efficient techniques for the processing of large classes of acyclic aggregate queries, we focus on the particularly challenging class of cyclic queries. To apply the standard approach of using decompositions to make cyclic queries acyclic successfully, it is clear that we require secondary optimisation objectives in addition to minimising the width of decompositions. To address this challenge, we introduce soft hypertree decompositions and the associated measure of soft hypertree width (*shw*). By avoiding the special condition of hypertree decompositions, we gain algorithmic flexibility while retaining the tractability of computing width-k decompositions. By developing an end-to-end pipeline, we achieve a seamless integration of these optimisations into DBMSs. Experimental evaluation on cyclic queries confirms that we can achieve significant improvements in decomposition quality, which translates to performance gains in practice.

Kurzfassung

Die Abarbeitung von Joins ist eine der grundlegenden Herausforderungen von Datenbanksystemen, insbesonders bei Zwischenergebnissen die groß werden. Dieses Problem tritt besonders häufig in analytischen Abfragen auf, welche Daten über viele Relationen aggregieren, aber nur ein relativ kleines Endergebnis produzieren. Obwohl die überwiegende Mehrheit von nutzergenerierten Abfragen azyklisch ist, und uns das fundamentale Ergebnis von Yannakakis eine Lösung bietet um in solchen Fällen unnötige Zwischenergebnisse zu vermeiden, ist solch eine strukturbasierte Optimierung von Abfragen bisher nicht in Standard-Datenbanksystemen zu finden. In dieser Dissertation arbeiten wir daran, diese Lücke zwischen Theorie und Praxis zu schließen.

Um die Vorteile und Einschränkungen von strukturbasierter Optimierung von Abfragen zu untersuchen, führen wir eine Implementierung von Yannakakis' Algorithmus ein, welche auf der Umschreibung von Abfragen basiert. Experimente zeigen, dass dieser Ansatz die Ausführung von schwierigen Instanzen deutlich beschleunigen kann, aber in den meisten Fällen keine Verbesserung erzielt. Zusätzlich identifizieren wir die Klasse 0MA (zero-materialisation answerable), von Anfragen, welche vollständig ohne Materialisierung von Zwischenergebnissen beantwortet werden kann. Diese Erkenntnisse motivieren die Formulierung dieses Problems als Entscheidung zwischen zwei Algorithmen. Um dieses Problem zu lösen wenden wir Machine Learning Methoden an, erstellen ein neues Datenset von Abfragen und zeigen, dass wir erfolgreich entscheiden können, wann die Umschreibung der Abfrage vorteilhaft ist.

Als nächstes erweitern wir die immer noch recht einschränkende Klasse 0MA, und führen *guarded* und *piecewise-guarded* Aggregatabfragen ein, wodurch wesentlich mehr Aggregatabfragen abgedeckt werden können während Materialisierung vermieden wird. Wir integrieren dies in Spark SQL, indem wir logische Optimierungen sowie einen neuen physischen Operator *AggJoin* einführen. Durch experimentelle Evaluierung bestätigen wir, dass die Implementierung die Ausführung von vielen Aggregatabfragen von Benchmarks beschleunigt aber nie einzelne verlangsamt. Um ebenfalls beliebige azyklische Aggregatabfragen, die über die Klasse *piecewise-guarded* hinausgehen, wo Materialisierung nicht vollständig vermieden werden kann, abzudecken, erweitern wir den *AggJoin* zum *GroupAggJoin*. Dadurch kann Materialisierung erheblich reduziert werden. Durch die Einführung eines neuen Benchmarks für solche Abfragen zeigen wir, dass unsere Methode klare Verbesserungen erzielt.

Nachdem wir effiziente Lösungen für die meisten azyklischen Aggregatabfragen präsentiert haben, widmen wir uns den besonders schwierigen zyklischen Abfragen. Der Standardansatz aus der Theorie ist, eine Zerlegung (decomposition) minimaler Width("Weite") der Abfrage zu berechnen und damit die zyklische Abfrage wie eine azyklische zu lösen. Allerdings hat sich gezeigt, dass es neben der Width weitere Optimierungsziele benötigt. Als Lösung für dieses Problem führen wir *soft hypertree decompositions* (eine allgemeinere Variante der Hypergraph-Zerlegung) ein sowie die assoziierte Width *soft hypertree width (shw)*. Durch die Definition erlangen wir mehr Flexibilität in der Berechnung der Zerlegung während das Problem k-Width Zerlegungen zu finden effizient lösbar bleibt. Durch die Entwicklung einer end-to-end Pipeline erreichen wir die reibungslose Integration in bestehende Systeme. Experimente auf zyklischen Benchmark-Instanzen zeigen, dass wir wesentliche Verbesserungen in der Qualität der Zerlegungen erreichen können.

Contents

CHAPTER 1

Introduction

In the first three sections of the introduction, we give a high-level and non-technical motivation of the challenges addressed in this thesis. In Section 1.4, we outline the main research questions and contributions.

This work was accomplished in collaboration with many colleagues, and resulted in several publications. Details on the publications and the correspondence of publications to the chapters of the thesis are provided in Section 1.4.

1.1 Large Intermediate Results in Join Queries

Join processing is a challenging task at the core of query engines, and a major factor in overall database performance. Nevertheless, despite decades of improvements in database systems, query engines still struggle when processing join queries in which the intermediate results become large. This situation is particularly common in analytical queries, which aggregate data over numerous relations, and reduce it to very few rows in the final output. Despite the well-known result by Yannakakis [162] providing us, in theory, with a solution for difficult acyclic queries exhibiting this problem, this approach or adaptations have not yet found their way into mainstream database systems. Such queries are becoming more and more common – for example, queries automatically generated by business intelligence tools may easily reach sizes where this problem is frequent [122]. It, therefore, should be a requirement for DBMSs today to cope with such queries. However, as we will see, the integration of Yannakakis' algorithm into systems, with wide applicability, while maintaining at least the original performance over all queries, is not straightforward.

Conjunctive Queries (CQs) correspond to SQL *SELECT-FROM-WHERE* expressions with conjunctions of equality conditions, and thus constitute an important class of queries. The traditional approach to evaluate a join query is to split it into a sequence

of two-way joins. One of the main tasks of query optimisation is then to determine the optimal or at least a good join order. In particular, part of finding a good join order is avoiding the costly computation of large intermediate results as far as possible. However, typical systems rely on some combination of heuristics and optimisation procedures to determine the join order for given queries. Hence, even for moderately large queries, the resulting optimisation problems become too difficult to solve exactly and the quality of the resulting join orders degrades quickly. For instance, PostgreSQL 14 by default performs a full search for the optimal plan only up to 11 joins before falling back on heuristic optimisation techniques. Sophisticated pruning methods and parallelisation have been shown to push this threshold higher [110, 109], but the task still remains fundamentally challenging. Moreover, the problem of huge intermediate results is not restricted to the choice of a bad join order.

Yannakakis' algorithm can be used to answer acyclic conjunctive queries (ACQs) – an important subclass of CQs covering most practical queries – efficiently. This is achieved by eliminating the blowup in the intermediate results, making use of the query's acyclic structure, and using semi-joins as opposed to joins, to eliminate dangling tuples which do not contribute to the final output. In theory, it seems clear that this approach is preferable to a sequence of binary joins. However, beside specialized research prototypes [2, 130, 152, 157], this approach has not seen adoption in standard DBMSs and query engines. Our overarching goal is, therefore, to make the evaluation of large classes of queries, which systems currently struggle with due to large intermediate results, feasible in mainstream DBMSs.

The first step towards this goal is to explore integrations of Yannakakis' algorithm into different DBMSs, and identify in which cases it proves to be beneficial to the performance. We thus implement an external rewriting which forces the DBMS to execute the query following Yannakakis' algorithm. In the course of this work, we identify the class *0MA* (definitions are provided in Chapter 4), consisting of acyclic queries followed by simple aggregations where the multiplicity of values does not matter (set-safety), such as `MIN` or `MAX`, over single relations (guardedness). For this class of queries, the Yannakakis-rewriting based approach provides significant improvements. Several new questions come up, motivating the rest of the work in this thesis. Still, due to the overhead of the rewriting, even on the very favourable 0MA class, there are cases where not even the baseline performance is achieved. On the enumeration queries, which require the full Yannakakis algorithm with its 3 traversals, and the materialisation of the entire output, performance is degraded in most cases as there is no great benefit from Yannakakis' algorithm due to the effort required to enumerate and materialise the final output in addition to the overhead introduced by the rewriting.

Having made these observations, it would clearly be desirable to reliably identify the cases where the Yannakakis-rewriting outperforms the standard join implementation.

We now consider an example in which a simple change in the query decides whether a Yannakakis rewriting is beneficial or not. In the query in Figure 1.1, Yannakakis-style evaluation is significantly faster (by a factor of ca. 30) than the original evaluation method

```
SELECT  MIN(c.Id)
FROM    comments AS c, posts AS p, votes AS v, users AS u
WHERE   u.Id = p.OwnerUserId AND u.Id = c.UserId AND
   u.Id = v.UserId AND u.Views>=0 AND p.Score>=0 AND
   p.Score<=28 AND p.ViewCount>=0 AND p.ViewCount<=6517
   AND p.AnswerCount>=0 AND p.AnswerCount<=5 AND
   p.FavoriteCount>=0 AND p.FavoriteCount<=8 AND
   c.CreationDate>='2010-07-27 12:03:40' AND
   p.CreationDate>='2010-07-27 11:29:20' AND
   p.CreationDate<='2014-09-13 02:50:15' AND
   u.CreationDate>='2010-07-27 09:38:05'
```

Figure 1.1: Slightly modified query 121-097 from the STATS benchmark: original runtime on PostgreSQL (3.38s) vs. Yannakakis-style evaluation (0.11s). After changing the filter condition to `p.FavoriteCount>=8`: original runtime on PostgreSQL (0.05s) vs. Yannakakis-style evaluation (0.09s).

of PostgreSQL. However, when we modify one of the conditions in the WHERE clause from ≥ 0 to ≥ 8 (which, together with the ≤ 8 condition, yields $= 8$), then suddenly the original evaluation method of PostgreSQL is faster.

In order to apply this version of Yannakakis-style query evaluation, we urgently need a method that decides when to apply the optimisation technique and when to stick to the original evaluation method of the database management system (DBMS). We therefore aim at developing such a decision procedure. We consider general acyclic queries (which join all relations after the semi-join-based elimination of dangling tuples), as well as 0MA queries (which only require a bottom-up traversal of the join tree with semi-joins).

To cover a diverse set of database technologies, we study three quite different DBMSs, namely PostgreSQL [146] (as a "classical" row-oriented relational DBMS), DuckDB [134] (as a column-oriented, in-memory database), and Spark SQL [164] (as a database engine specifically designed for distributed data processing in a cluster environment).

We treat the design of a decision procedure between different query evaluation techniques as an *algorithm selection problem*, that we want to solve by applying Machine Learning (ML) techniques. In recent years, ML techniques have been frequently applied to solve database problems – above all cardinality estimation and join order optimisation (see, e.g., the survey paper [170]). Our focus, however, is different: we are interested in the overall runtime of query evaluation under different evaluation methods. We ultimately aim to improve end-to-end runtimes over entire workloads, rather than local optimisations of some specific subtask of query evaluation.

To approach this algorithm selection as a classification problem, we need to select queries and data from benchmarks, and identify relevant features that characterize these queries. Together with the runtimes obtained on our target DBMSs, this forms our training data. SQL benchmarks generally aim at testing specific features of DBMSs and are relatively small for training and testing ML models. We therefore create a new dataset by adapting and combining several benchmarks followed by data augmentation, to achieve the variety

and size of the dataset that makes it suitable for model training.

1.2 Efficient Processing of Acyclic Aggregate Queries

The techniques introduced in Section 1.1 and later covered in Chapter 4 allow us to apply Yannakakis' algorithm quite reliably, whenever it is expected to yield an improvement, for 0MA queries, and for enumeration queries. However, we also observe that on the vast majority of enumeration queries, the query performance is degraded. Especially in aggregate queries, where only a limited amount of information is ultimately part of the output, it would be highly desirable to reduce or, ideally, avoid altogether the materialisation of intermediate join results.

Several works [132, 51] investigated how variations of the same algorithmic idea also apply to join queries with `COUNT` aggregates. Subsequently, these ideas were extended to more general aggregate queries in the FAQ-framework (Functional Aggregate Queries) [91] and, similarly, under the name AJAR (Aggregations and Joins over Annotated Relations) [90]. We offer a more detailed account of related work in Chapter 3.

However, previous works in this area have left a gap: Most approaches (such as FAQ and AJAR mentioned above) aim at *reducing* (not eliminating) the number of joins and/or the cost of computing them by applying sophisticated join techniques. But the computation and materialisation of joins remains the dominating cost factor. On the other hand, approaches that avoid the computation or materialisation of intermediate join results depend on severe restrictions of the class of queries, such as Boolean queries or 0MA queries. Indeed, as we will show in our empirical evaluation in Section 5.3, only a small fraction of the queries in the standard benchmarks considered here satisfies these restrictions.

Our goal is thus to identify a class of aggregate queries which can be evaluated without the need to compute or materialise any joins and which, nevertheless, cover many practical cases. The key to this class of queries is the notion of *guardedness*. More specifically, we call a query with aggregates on top of an acyclic join query *guarded* if all attributes involved in a `GROUP BY` clause and in any aggregate expression are contained in a single relation, referred to as the "*guard*". Here we allow any aggregate function from the ANSI SQL standard – including statistical functions such as `MEDIAN`, `VARIANCE`, `STDDEV`, `CORR`, etc.

Example 1.2.1. *We illustrate the basic ideas with the simple query given in Figure 1.2 over the well-known TPC-H schema. The query asks for the median of the account balance of suppliers from one of the regions 'Europe' or 'Asia' for parts with above average price. The join-structure of the query is clearly acyclic as is witnessed by the join tree displayed in Figure 1.2. Moreover the query is trivially* guarded*, since it has no grouping and the aggregation is over a single attribute.*

Note that the subquery is only used to realise a selection (locally) on the relation `part`*. After applying this selection on the* `part` *relation and also the selection on the* `region`

```
SELECT MEDIAN(s_acctbal)
FROM part, partsupp, supplier,
  nation, region
WHERE p_partkey   = ps_partkey
  AND s_suppkey   = ps_suppkey
  AND n_nationkey = s_nationkey
  AND r_regionkey = n_regionkey
  AND p_price >
   (SELECT avg (p_price) FROM part)
AND r_name IN ('Europe', 'Asia')
```

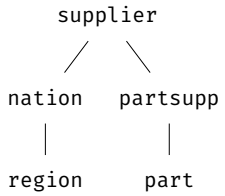


Figure 1.2: Query over the TPC-H schema and its corresponding join tree

relation, the query can be evaluated by propagating frequencies *of attribute value combinations rather than intermediate join results up the join tree. The* `MEDIAN`*-aggregate can then be evaluated on the resulting relation at the root node. Indeed, suppose that we have computed all tuples* $t_1, \ldots, t_n$ *of relation* `supplier` *together with the corresponding frequencies* $c_1, \ldots, c_n$ *of these tuples in the full join result of the five relations. Then we can order the values* $v_1, \ldots, v_n$ *of these tuples in ascending order and, by taking the frequencies* $c_1, \ldots, c_n$ *into account, it is an easy task to read off the median value. This is in sharp contrast to traditional query evaluation, which would first compute the* join of the five relations *and evaluate the aggregate on the full join result.* ◇

As we will see in Section 5.3, all the queries in the STATS-CEB [76] and SNAP [102] benchmarks are thus covered, and so is a small number of queries in the other benchmarks studied here. However, for the most commonly used aggregate functions `MIN`, `MAX`, `COUNT`, `SUM`, and `AVG`, the guardedness restriction can be significantly relaxed. We thus define *piecewise-guarded* queries as queries with aggregates on top of an acyclic join query, where the attributes in a `GROUP BY` clause and the attributes jointly occurring in an aggregate expression *each* are contained in a single relation. That is, the `GROUP BY` clause and also each aggregate expression has a *guard*, but all these guards may be different. It will turn out that, with this relaxed restriction, we can cover all JOB [101] queries and a significant number of queries in the TPC-H [150] and TPC-DS [149] benchmarks.

We will show how to realise Yannakakis-style evaluation for guarded and piecewise-guarded queries by rewriting subtrees in the logical query plan. In this process, joins are either replaced by semi-joins or they are immediately followed by aggregation. Importantly, in the latter case, the number of tuples to be propagated up the join tree is the same as in the case of semi-joins. The only extension needed is to add columns for the total frequencies (corresponding to `COUNT(*)`) and for the various aggregate expressions contained in the query. All of this can, in a very natural way, be implemented as part of the logical optimisation step. This approach thus applies also to subqueries and automatically works in conjunction with other optimisation techniques such as subquery decorrelation. An additional benefit of this optimisation is that it requires no cost-based optimisation, making it particularly attractive for systems with a limited or no cost model, such as Spark SQL.

As a further optimisation, we introduce a new physical operator that intuitively implements a semi-join that keeps track of frequencies and other aggregate values, and which can be implemented through minimal changes to standard join algorithms (see Section 5.2). It thus integrates smoothly in any typical SQL execution engine. We integrate both the logical optimisation and the new physical operator into Spark SQL, which was specifically designed to cope with complex analytical queries. The performance gain observed in our experimental evaluation on several standard benchmarks can reach up to one or two orders of magnitude for analytical queries involving aggregates on top of non-trivial join or path queries. Notably, our method incurs no performance degradation even for simple queries where the size of intermediate results never gets too big anyway.

To go even beyond piecewise-guarded queries and towards arbitrary aggregates over CQs, where materialisation sometimes cannot be avoided, we extend the implementation and introduce another physical operator: *GroupAggJoin*, which brings together grouping and aggregation in a semi-join-like operator (see Section 5.4). This enables the propagation of grouping attributes, and significantly reduces the amounts of materialisation required. For the evaluation of this extension, we require a benchmark focusing on unguarded queries, and therefore introduce a new benchmark based on the Join-Order-Benchmark [101], from which we derive unguarded queries of various degrees of unguardedness. Results on this benchmark are very promising, showing that this approach achieves clear speedups on moderately-unguarded queries.

1.3 Towards Practical Solutions for Cyclic Query Processing

Since their introduction over 25 years ago, hypertree decompositions (HDs) and hypertree width (*hw*) have emerged as a cornerstone in the landscape of database research. Their enduring relevance lies in their remarkable ability to balance theoretical elegance with practical applicability. Hypertree width generalises α-acyclicity, a foundational concept for the efficient evaluation of conjunctive queries (CQs) [162]. Indeed, CQs of bounded hypertree width can be evaluated in polynomial time. Crucially, determining whether a CQ has $hw \leq k$ for fixed $k \geq 1$ and, if so, constructing an HD of width $\leq k$ can also be achieved in polynomial time [62].

Over the decades, hypertree width has inspired the development of generalised hypertree width (*ghw*) [64] and fractional hypertree width (*fhw*) [73], each extending the boundaries of tractable CQ evaluation. These generalisations induce larger classes of tractable CQs. Notably, the hierarchy

$$fhw(q) \leq ghw(q) \leq hw(q)$$

holds for any CQ q, with the respective width measure determining the exponent of the polynomial bound on the query evaluation time. Consequently, practitioners naturally gravitate towards width measures that promise the smallest possible decomposition widths. Despite this motivation, HDs and *hw* retain a crucial distinction: *hw* is, to date,

the most general width measure for which it can be decided in polynomial time [62] if the width of a given query is below some fixed bound k. By contrast, generalised and fractional hypertree width, while often yielding smaller width values, suffer from computational intractability in their exact determination [64, 60]. We introduce a novel width notion—soft hypertree width (*shw*) – that overcomes this trade-off by retaining polynomial-time decidability while potentially reducing the width compared to *hw*.

But this is only a first step towards a larger goal. Query evaluation based on decomposition methods works via a transformation of a given CQ with low width into an acyclic CQ by computing "local" joins for the bags at each node of the decomposition. The acyclic CQ is then evaluated by Yannakakis' algorithm [162]. From a theoretical perspective, the complexity of this approach solely depends on the width, while the actual cost of the local joins and the size of the relations produced by these joins is ignored. To remedy this short-coming, Scarcello et al. [136] introduced *weighted HDs* to incorporate costs into HDs. More precisely, among the HDs with minimal width, they aimed at an HD that minimises the cost of the local joins needed to transform the CQ into an acyclic one and the cost of the (semi-)joins required by Yannakakis' algorithm. Promising first empirical results were obtained with this approach.

However, there is more to decomposition-based query optimisation than minimising the width. For instance, $hw = 2$ means that we have to compute joins of two relations to transform the given query into an acyclic one. Regarding width, it makes no difference if the join is a Cartesian product of two completely unrelated relations or a comparatively cheap join along a foreign key relationship. This phenomenon is, in fact, confirmed by Scarcello et al. [136] when they report on decompositions with higher width but lower cost in their experimental results. Apart from the pure join costs at individual nodes or between neighbouring nodes in the decomposition, there may be yet other reasons why a decomposition with slightly higher width should be preferred. For instance, in a distributed environment, it makes a huge difference, if the required joins and semi-joins are between relations on the same server or on different servers. In other words, apart from incorporating costs of the joins at a node and between neighbouring nodes in the HD, a more general approach that considers constraints and preferences is called for.

We propose a novel framework based on the notion of *candidate tree decompositions* (CTDs) from [60]. A CTD is a tree decomposition (TD) whose bags have to be chosen from a specified set of vertex sets (= the *candidate bags*). One can then reduce the problem of finding a particular decomposition (in particular, generalised or fractional hypertree decomposition) for a given CQ q to the problem of finding a CTD for an appropriately defined set of candidate bags. In [60], this idea was used (by specifying appropriate sets of candidate bags) to identify tractable fragments of deciding, for given CQ q and fixed k, if $ghw(H) \leq k$ or $fhw(H) \leq k$ holds. However, as was observed in [60], it is unclear how the idea of CTDs could be applied to *hw*-computation, where parent/child relationships between nodes in a decomposition are critical.

We will overcome this problem by introducing a relaxed notion of HDs and *hw*, which we will refer to as *soft HDs* and *soft hw* (*shw*). The crux of this approach will be a

relaxation of the so-called "special condition" of HDs (formal definitions of all concepts mentioned in the introduction will be given in Chapter 2): it will be restrictive enough to guarantee a polynomial upper bound on the number of candidate bags and relaxed enough to make the approach oblivious of any parent/child relationships of nodes in the decomposition. We thus create a novel framework based on CTDs, that paves the way for several improvements over the original approach based on HDs and *hw*, while retaining the crucial tractability of deciding, for fixed k, if the width is $\leq k$ and, if so, computing a decomposition of the desired width.

Importantly, *shw* constitutes an improvement in terms of the width, i.e., it is never greater than *hw* and there exist CQs q with strictly smaller *shw* than *hw*. Basing our new width notion *shw* on CTDs gives us a lot of computational flexibility. In particular, in [60], a straightforward algorithm was presented for computing a CTD for a given set of candidate bags. We will show how various types of constraints (such as disallowing Cartesian products) and preferences (which includes the minimisation of costs functions in [136] as an important special case) can be incorporated into the computation of CTDs without destroying the polynomial time complexity. We emphasise that even though our focus in this work is on soft HDs and *shw*, the incorporation of constraints and preferences into the CTD computation equally applies to any other use of CTDs – such as the above mentioned computation of generalised and fractional hypertree decomposition in certain settings proposed in [60]. We carry out first experimental results with a prototype implementation of our approach. In short, basing query evaluation on soft HDs and incorporating constraints and preferences into their construction may indeed lead to significant performance gains. Moreover, the computation of soft HDs in a bottom-up fashion via CTDs instead of the top-down construction [54, 66] or parallel computation of HDs [58] is in the order of milliseconds and does not create a new bottleneck.

1.4 Research Questions and Results

1.4.1 Integrating Yannakakis' Algorithm into Systems

The results presented in this subsection, as well as the corresponding content of Chapter 4, is based on joint work with Daniela Böhm, Georg Gottlob, Matthias Lanzinger, Davide Longo, Cem Okulmus, and Reinhard Pichler. It is mainly based on (at the time of writing) unpublished work:

- [57] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, Alexander Selzer. Structure-Guided Query Evaluation: Towards Bridging the Gap from Theory to Practice.
- [131] Daniela Böhm, Georg Gottlob, Matthias Lanzinger, Davide M. Longo, Cem Okulmus, Reinhard Pichler, Alexander Selzer. Selective Use of Yannakakis' Algorithm to Improve Query Performance: Machine Learning to the Rescue.

A short version of [57] was presented at the Alberto Mendelzon Workshop:

- [56] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, Alexander Selzer. Reaching Back to Move Forward: Using Old Ideas to Achieve a New Level of Query optimisation (short paper). 15th Alberto Mendelzon International Workshop on Foundations of Data Management. 2023.

As a starting point on the path towards bringing structure-guided query processing into systems, we begin by exploring the potential and possible shortcomings of such an approach.

> **Research Question I:** To what extent can Yannakakis' algorithm be integrated into database systems with performance benefits?

In our first exploratory study (Chapter 4), we choose the path of implementing a SQL rewriting system for expressing Yannakakis' algorithm – this has the advantage of being system-agnostic and allows us to test the approach on three very different systems: PostgreSQL, SparkSQL, and DuckDB.

> **Main Result 1:** We develop a lightweight rewriting based implementation of Yannakakis' algorithm and integrate it into 3 DBMSs.

In order to investigate the impact of the rewriting on particularly challenging join queries, we evaluate it on a benchmark by Mancini et al. [110] over the MusicBrainz dataset [1] on the three systems. The results show that the approach outperforms the system alone on many hard instances but at the same time adds significant overhead to query execution, leading to losses in performance on most simple queries.

> **Main Result 2:** Experimental evaluation shows that the Yannakakis-style query evaluation leads to performance gains on many hard instances but degrades performance on most simple instances.

As the standard variant of Yannakakis' algorithm for enumeration queries requires two traversals of semi-joins over the join tree followed by an expensive expansion and materialisation of the full result, we introduce the class of zero-materialisation answerable (0MA) queries. For 0MA queries, it is only required to perform the inexpensive bottom-up traversal of semi-joins. Experiments confirm the effectiveness of the rewriting-based approach for 0MA queries.

> **Main Result 3:** We identify and define the class of zero-materialisation answerable (0MA) queries and show experimentally that instances of this class can in general be evaluated very efficiently using Yannakakis-style query processing.

The mixed positive as well as negative results of the rewriting-based approach towards Yannakakis-style query execution motivate us to improve on it. Since the rewriting is highly effective on many instances but disappointing on most, we can consider the decision whether to apply it as an algorithm selection problem. Hence, we need to be able to predict whether the rewriting is expected to yield an improvement.

> **Research Question II:** Is it possible to construct a reliable decision procedure for deciding when to apply Yannakakis-style execution effectively?

To solve the algorithm selection problem, we apply machine learning techniques. For effectively training the models, we however require a large-enough dataset of training data – larger than what standard benchmarks offer. In order to obtain a sufficiently large data set, we combine data from 5 benchmarks [101, 76, 103, 116, 80] and augment the 219 queries from the benchmarks in order to generate 2936 0MA queries and 1741 enumeration queries in total.

> **Main Result 4:** We introduce a new dataset and benchmark (*MEAMBench*) by combining and extending several benchmarks, in order to support machine learning tasks.

After evaluating several machine learning models on the new dataset, we conclude that a simple decision tree classification or regression model quite effectively learns to decide when to perform the rewriting.

> **Main Result 5:** We present a decision procedure for reliably deciding when to perform Yannakakis-style execution.

1.4.2 Efficient Join-Aggregate Processing

The results presented in this subsection, as well as the corresponding content of Chapter 5 is based on joint work with Matthias Lanzinger and Reinhard Pichler:

- [100] Matthias Lanzinger, Reinhard Pichler, Alexander Selzer. Avoiding Materialisation for Guarded Aggregate Queries. VLDB 2025

A preliminary version of this work was also presented at the Alberto Mendelzon Workshop:

- [98] Matthias Lanzinger, Reinhard Pichler, Alexander Selzer. Avoiding Materialisation for Guarded Aggregate Queries. 16th Alberto Mendelzon International Workshop on Foundations of Data Management. 2024.

The extension to unguarded aggregate queries is, at the time of writing, novel work.

While demonstrating the potential of Yannakakis-style query evaluation via query rewriting, and mitigating the drawbacks via a decision procedure, the previous results also highlight a major gap which we seek to address next: Although 0MA queries can be solved efficiently, the standard version of Yannakakis' algorithm is not optimal for the general class of enumeration queries and by extension also the very important class of aggregate queries. Aggregate queries are usually expressed as an aggregation (and possibly grouping) operation after an enumeration query. However, analytical aggregate queries, while commonly joining a large number of tables, tend to produce relatively few output rows. This results in a significant materialisation of intermediate results. Clearly, there is a need to identify classes of queries broader than 0MA and go beyond the standard version of Yannakakis' algorithm.

> **Research Question III:** Are there broader classes of aggregate queries for which we can avoid materialising the intermediate join result?

Our first step towards extending the class of 0MA queries, which is restricted by the set-safety and guardedness conditions, is to remove one of the two conditions: set-safety. This means that we now allow arbitrary aggregate functions, removing the restriction to only aggregate functions which are unaffected by duplicate values (such as MIN/MAX). To enable this, we make use of an extension of Yannakakis' algorithm for counting [132], keeping track of tuple frequencies, which allow us to reconstruct arbitrary aggregate functions.

> **Main Result 4:** We identify the class of *guarded* aggregate queries.

Next, we significantly weaken the remaining 0MA condition: guardedness. We eliminate the requirement for all output attributes to be present in the root node of the join tree by propagating the results of the aggregate functions themselves up the tree. This approach incurs minimal overhead by only adding additional columns to relations of the join tree.

> **Main Result 5:** We extend the class of guarded aggregate queries to *piecewise-guarded* aggregate queries.

Since Yannakakis-style query processing has not been adopted by mainstream DBMSs, efficiently integrating it into existing query optimisers remains an open challenge.

> **Research Question IV:** How can we integrate Yannakakis-style query processing into a query optimiser with minimal to no overhead?

To perform Yannakakis-like bottom-up propagation of frequencies and aggregates, we have to extend Spark SQL's optimiser to, first of all, detect applicable queries, i.e., logical (sub)plans, and then rewrite these to follow the new execution strategy.

> **Main Result 6:** We implement the logical optimisations by extending Spark SQL's optimiser.

Experiments show that the purely logical optimisation is effective in most situations, but introduces some overhead due to the lack of a real semi-join-like operator and therefore the need for joins followed by aggregation. Thus, to practically eliminate the overhead of the optimisation, we introduce a new physical operator performing the aggregation directly inside of the join loop: *AggJoin*. In accordance to Spark SQL's standard physical operators – SortMergeJoin, BroadcastHashJoin, and ShuffledHashJoin – 3 variants of the AggJoin are implemented.

> **Main Result 7:** We introduce and integrate a novel semi-join-like physical operator combining the join and aggregation operations: *AggJoin*.

In order to assess both the applicability and the performance of the optimisations, we evaluate the modified version of Spark SQL against the original version, on 5 standard benchmarks. Results show that the optimisation achieves modest up to very significant speedups on most queries while performance is never degraded.

> **Main Result 8:** Experimental evaluation over 5 benchmarks shows wide applicability of the optimisations and performance gains without noticeable tradeoffs.

Piecewise-guarded queries already cover the majority of queries in the benchmarks we consider here. To cover arbitrary aggregations over ACQs, the only remaining restriction to weaken or remove is piecewise-guardedness.

> **Research Question V:** Can we extend these techniques to go beyond piecewise-guarded queries efficiently?

Unguarded queries – which require us to compute the enumeration of at least two different relations' attributes – essentially leave us with no other choice than to compute joins and to materialise some intermediate results. We can therefore no longer avoid all materialisation but we can make efforts to keep it minimal. A natural extension of the approach we have considered up to now is to integrate grouping into the AggJoin operator: in the join loop, we now perform grouping as well as aggregation. This enables the propagation of only the unguarded attributes to the root of the tree, keeping materialisation low.

Main Result 9: We further extend the physical *AggJoin* operator to the *GroupAggJoin* operator and integrate it into Spark SQL.

To the best extent of our knowledge, there are currently no benchmarks focusing on unguarded queries. Therefore, we construct a new benchmark based on the Join-Order-Benchmark (JOB): JOBUnguarded. We adapt the existing (piecewise-guarded) JOB queries and, for each query, create up to 8 unguarded variants, starting from 2 unguarded attributes up to 9.

Main Result 10: We introduce a new benchmark for unguarded queries based on the Join-Order-Benchmark (JOB): JOBUnguarded.

Benchmarks on JOBUnguarded show that the GroupAggJoin-based implementation outperforms Spark SQL on the vast majority of slightly-unguarded queries (2-5 unguarded attributes). Even on strongly unguarded queries (9 unguarded attributes), our optimisation achieves speedups on the majority of queries and is competitive with the standard implementation of Spark SQL.

Main Result 11: Experimental evaluation of the GroupAggJoin-based implementation shows promising results for unguarded queries.

1.4.3 Cyclic Join Queries – From Theory to Practice

The results presented in this subsection, as well as the corresponding content of Chapter 6 is based on joint work with Georg Gottlob, Matthias Lanzinger, Cem Okulmus, and Reinhard Pichler:

- [97] Matthias Lanzinger, Cem Okulmus, Reinhard Pichler, Alexander Selzer, Georg Gottlob. Soft and Constrained Hypertree Width. PODS 2025

So far, we have already shown that structure-guided, i.e., Yannakakis-style query processing can be integrated into systems with great benefits for almost all acyclic aggregate queries. In the final Chapter 6, we make steps towards extending these techniques to apply to CQs in general. In preliminary experiments described briefly in Chapter 4, we identify a major challenge in cyclic query processing: the standard approach of computing a minimal-width decomposition in order to obtain a join tree and apply Yannakakis-style execution fails to take into consideration the real-world costs of performing specific joins. Standard optimisers even tend to outperform this approach despite the suboptimal execution strategy since they are able to mitigate the problem via cost-based optimisation.

> **Research Question VI:** How can we incorporate constraints and cost functions into the search for optimal decompositions?

We make use of the concept of *candidate tree decompositions* (CTDs) from [60] – tree decompositions whose bags have to be chosen from a set of candidate bags. It was observed in [60], that it is unclear how the idea of CTDs could be applied to *hw*-computation, where parent/child relationships between nodes in a decomposition are critical. We will overcome this problem by introducing a relaxed notion of HDs and *hw*, which we will refer to as *soft HDs* and *soft hw* (*shw*). The key idea behind this approach is a relaxation of the "special condition" of HDs. While retaining the important tractability of, for fixed k, if the width is $\leq k$, computing a decomposition of the desired width, we gain the additional flexibility of incorporating constraints and preferences (including cost funnctions) into the computation of CTDs while retaining polynomial time complexity.

> **Main Result 12:** We introduce a relaxation of HDs and *hw*: *soft HDs* and *soft hw (shw)*, offering more computational flexibility, which allows us to incorporate contraints and preferences into the search for decompositions.

To integrate this approach into systems, we make use of the idea of the Yannakakis-rewriting from Chapter 4, and essentially extend it to perform Yannakakis' algorithm along a decomposition instead of a join tree. Furthermore, to be able to set up cost functions, we are now in need of statistical information about the query and the database, for which the component is also extended. We implement a component for the computation of

decompositions while considering constraints and preferences. Putting it all together, we obtain an end-to-end pipeline allowing for the fully automated optimisation of cyclic queries.

> **Main Result 13:** An end-to-end system is implemented for extracting costs from a database, to computing optimal decompositions, and performing a Yannakakis-style rewriting.

We select cyclic queries from several benchmarks and perform experiments to test the performance of the optimisation pipeline. The results show that both the restriction of decompositions through constraints as well as the use of cost functions can greatly improve the quality of decompositions.

> **Main Result 14:** Experimental evaluation shows that this approach is promising and that including costs and constraints into the search can be effective to find good decompositions in practice.

1.5 Structure of the Thesis

In Chapter 2 we provide preliminary definitions and results assumed in the main chapters of the thesis. We describe relevant related work in Chapter 3. The next chapters 4, 5, and 6 constitute the main part of the thesis. These chapters cover the main results described in Section 1.4 in detail. We conclude the thesis in Chapter 7, summarising the main results and looking into further open research challenges as future work.

CHAPTER 2

Preliminaries

Conjunctive Queries and Beyond. The basic form of queries studied here are Conjunctive Queries (CQs), which correspond to select-project-join queries in Relational Algebra. It is convenient to consider CQs as Relational Algebra expressions of the form $Q = \pi_U(R_1 \bowtie \ldots \bowtie R_n)$. Here we assume w.l.o.g., that equi-joins have been replaced by natural joins via appropriate renaming of attributes. Moreover, we assume that selections applying to a single relation have been pushed immediately in front of this relation and the R_i's are the result of these selections. The projection list U consists of attributes occurring in the R_i's.

To go beyond CQs, we will also consider the extension of Relational Algebra that applies aggregates on top of ACQs and that may contain "arbitrary" selections applied to single relations (that is, not only equality conditions, as is usually assumed for CQs [3]). Moreover, we allow grouping, which can also take care of the projection. In other words, we are interested in queries of the form

$$Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)](R_1 \bowtie \cdots \bowtie R_n) \tag{2.1}$$

where $\gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]$ denotes the grouping operation for attributes $g_1, \ldots, g_\ell$ and aggregate expressions $A_1(a_1), \ldots, A_m(a_m)$ for some (standard SQL[1]) aggregate functions $A_1, \ldots, A_m$ applied to expressions $a_1, \ldots, a_m$. The grouping attributes $g_1, \ldots, g_\ell$ are attributes occurring in the relations $R_1, \ldots, R_n$ and $a_1, \ldots, a_m$ are expressions formed over the attributes from $R_1, \ldots, R_n$. A simple query of the form shown in Equation (2.1) is given in Figure 1.1 (in SQL-syntax), together with a possible join tree of this query.

In Chapter 5, it will be convenient to use the following notation: suppose that we want to assign the result of a query Q of the form according to Equation (2.1) to a relation S

[1] i.e., ANSI standard

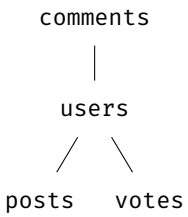


Figure 2.1: Join tree for the query in Fig. 1.1

with attributes $g_1, \ldots, g_\ell, C_1, \ldots, C_m$), such that the values of each aggregate expression $A_i(a_i)$ is assigned to the attribute C_i, then we will write

$$S := \gamma[g_1, \ldots, g_\ell,\ C_1 \leftarrow A_1(a_1), \ldots, C_m \leftarrow A_m(a_m)] \\ (R_1 \bowtie \cdots \bowtie R_n) \tag{2.2}$$

Acyclic Queries. An *acyclic conjunctive query* (an ACQ, for short) is a CQ $Q = \pi_U(R_1 \bowtie \ldots \bowtie R_n)$ that has a *join tree*, i.e., a rooted, labelled tree $\langle T, r, \lambda \rangle$ with root r, such that (1) λ is a bijection that assigns to each node of T one of the relations in $\{R_1, \ldots, R_n\}$ and (2) λ satisfies the so-called *connectedness condition*, i.e., if some attribute A occurs in both relations $\lambda(u_i)$ and $\lambda(u_j)$ for two nodes u_i and u_j, then A occurs in the relation $\lambda(u)$ for every node u along the path between u_i and u_j. Deciding if a CQ is acyclic and, in the positive case, constructing a join tree can be done very efficiently by the GYO-algorithm (named after the authors of [68, 163]). The join query underlying the SQL query in Figure 1.1 can be easily seen to be acyclic. A possible join tree is shown in Figure 2.1.

Yannakakis' Algorithm. In [162], Yannakakis showed that ACQs can be evaluated in time $O((||D|| + ||Q(D)||) \cdot ||Q||)$, i.e., linear w.r.t. the size of the input and output data and w.r.t. the size of the query. This bound applies to both, set and bag semantics. Let us ignore grouping, aggregation, and projection for a while and consider an ACQ Q of the form $R_1 \bowtie \ldots \bowtie R_n$ with join tree $\langle T, r, \lambda \rangle$. Yannakakis' algorithm (no matter whether we consider set or bag semantics) consists of a preparatory step followed by 3 traversals of T:

In the *preparatory step* we associate with each node u in the join tree T the relation $\lambda(u)$. If the CQ originally contained selection conditions on attributes of relation $\lambda(u)$, then we can now apply this selection. The 3 traversals of T consist of (1) a bottom-up traversal of semi-joins, (2) a top-down traversal of semi-joins, and (3) a bottom-up traversal of joins. Formally, let u be a node in T with child nodes $u_1, \ldots, u_k$ of u and let relations R, $R_{i_1}, \ldots, R_{i_k}$ be associated with the nodes u, $u_1, \ldots, u_k$ at some stage of the computation. In the 3 traversals (1), (2), and (3), respectively, they are modified as follows:

(1) $R = (((R \ltimes R_{i_1}) \ltimes R_{i_2}) \dots) \ltimes R_{i_k}$,

(2) $R_{i_j} = R_{i_j} \ltimes R$ for every $j \in \{1, \dots, k\}$, and

(3) $R = (((R \bowtie R_{i_1}) \bowtie R_{i_2}) \dots) \bowtie R_{i_k}$

The result of the query is the final relation associated with the root node r of T. Grouping and the evaluation of aggregates can be carried out as post-processing *after* the evaluation of the join query. In contrast, projection π_U can be integrated *into* this algorithm by projecting out in the second bottom-up traversal all attributes that neither occur in U nor further up in T. Attributes neither occurring in U nor in any join condition are projected out as part of the preparatory step.

The correctness of Yannakakis' algorithm is seen by a closer look at the relations resulting from each traversal of T. For a node u of T, let R denote the original relation associated with u, i.e., $\lambda(u) = R$, and let $R_{i_1}, \dots, R_{i_\ell}$ denote the relations labelling the nodes in the subtree T_u of T rooted at u. Moreover, let R' denote the relation resulting from each traversal of the join tree. We again write (1), (2), (3) to denote the 3 traversals of the join tree. Then it holds:

after (1), we have $R' = \pi_{Att(u)}(R_{i_1} \bowtie \dots \bowtie R_{i_\ell})$,

after (2), we have $R' = \pi_{Att(u)}(R_1 \bowtie \dots \bowtie R_n)$,

after (3), we have $R' = \pi_{Att(T_u)}(R_1 \bowtie \dots \bowtie R_n)$.

Here, we write $Att(u)$ and $Att(T_u)$ to denote the attributes of the relation $\lambda(u)$ and the attributes occurring in any of the relations $R_{i_1}, \dots, R_{i_\ell}$ labelling a node in T_u, respectively.

Hypergraphs. Hypergraphs have proved to be a useful abstraction of CQs and CSPs. Recall that a *hypergraph* H is a pair $(V(H), E(H))$ where $V(H)$ is called the set of vertices and $E(H) \subseteq 2^{V(H)}$ is the set of *(hyper)edges*. Then the hypergraph $H(\phi)$ corresponding to a CQ or CSP given by a logical formula ϕ has as vertices the variables occurring in ϕ. Moreover, every collection of variables jointly occurring in an atom of ϕ forms an edge in $H(\phi)$.

We write $I(v)$ for the set $\{e \in E(H) \mid v \in e\}$ of edges incident to vertex v. W.l.o.g., we assume that hypergraphs have no isolated vertices, i.e., every $v \in V(H)$ is in some edge. A set of vertices $U \subseteq V(H)$ induces the *induced subhypergraph* $H[U]$ with vertices U and edges $\{e \cap U \mid e \in E(H)\} \setminus \{\emptyset\}$.

Let H be a hypergraph and let u, v be two distinct vertices in H. A path from u to v is a sequence of vertices $w_1, \dots, w_m$ in H, such that $w_0 = u$, $w_m = v$, and, for every $j \in \{0, \dots, m-1\}$, the vertices w_j, w_{j+1} jointly occur in some edge of H. Now let $S \subseteq V(H)$. We say that two vertices u, v are $[S]$-*connected* if they are not in S and there is a path from u to v without any vertices of S. Two edges e, f are $[S]$-connected, if there are $[S]$-connected vertices $u \in e, v \in f$. An $[S]$-*component* is a maximal set of pairwise $[S]$-connected edges. For a set of edges $\lambda \subseteq E(H)$, we will simply speak of

$[\lambda]$-components in place of $[\bigcup\lambda]$-components. We note that the term "$[S]$-connected" is slightly misleading, since – contrary to what one might expect – it means that the vertices of S are actually *disallowed* in a path that connects two vertices or edges. However, this terminology has been commonly used in the literature on hypertree decompositions since their introduction in [62]. To avoid confusion, we have decided to stick to it also here.

Decompositions A *tree decomposition* (TD) of hypergraph H is a tuple (T, B) where T is a tree and $B : V(T) \to 2^{V(H)}$ such that the following hold:

1. for every $e \in E(H)$, there is a node u in T such that $e \subseteq B(u)$,
2. and for every $v \in V(H)$, $\{u \in V(T) \mid v \in B(u)\}$ induces a non-empty subtree of T.

The vertex sets $B(u)$ are referred to as *bags* of the TD. The latter condition is referred to as the *connectedness condition*. We sometimes assume that a TD is rooted. In this case, we write T_u for $u \in V(T)$ to refer to the subtree rooted at u. For a subtree T' of T, we write $B(T')$ for $\bigcup_{u \in V(T')} B(u)$.

A *generalised hypertree decomposition* (GHD) of a hypergraph H is a triple (T, λ, B), such that (T, B) is a tree decomposition and $\lambda\colon V(T) \to 2^{E(H)}$ satisfies $B(u) \subseteq \bigcup\lambda(u)$ for every $u \in V(T)$. That is, every $\lambda(u)$ is an *edge cover* of $B(u)$. A *hypertree decomposition* (HD) of a hypergraph H is a GHD (T, λ, B), where the tree T is rooted, and the so-called *special condition* holds, i.e., for every $u \in V(T)$, we have $B(T_u) \cap \bigcup\lambda(u) \subseteq B(u)$. Actually, in the literature on (generalised) hypertree decompositions, it is more common to use χ instead of B. Moreover, rather than explicitly stating B, χ, and λ as functions, they are usually considered as *labels* – writing B_u, λ_u, and χ_u rather than $B(u)$, $\chi(u)$, and $\lambda(u)$, and referring to them as B-, χ-, and λ-label of node u.

All these decompositions give rise to a notion of *width*: The width of a TD (T, B) is defined as $\max(\{|B(u)| : u \text{ is a node in } T\} - 1$, and the width of a (G)HD (T, λ, χ) is defined as $\max(\{|\lambda(u)| : u \text{ is a node in } T\}$. Then the *treewidth* $tw(H)$, the *hypertree-width* $hw(H)$, and the *generalised hypertree-width* $ghw(H)$ of a hypergraph H are defined as the minimum width over all TDs, HDs, and GHDs, respectively, of H. The problems of finding the $tw(H)$, $hw(H)$, or $ghw(H)$ for given hypergraph H (strictly speaking, the decision problem of deciding whether any of these width notions is $\leq k$ for given k) are NP-complete [12, 67]. Deciding $tw \leq k$ or $hw(H) \leq k$ becomes tractable, if k is a fixed constant. In contrast, deciding $ghw(H) \leq k$ remains NP-complete even if we fix $k = 2$ [64, 60].

We note that, in the literature on tree decompositions, it is common practice to define TDs for graphs (rather than hypergraphs) in the first place. The TDs of a hypergraph $H = (V(H), E(H))$ are then defined as TDs of the so-called *Gaifman graph* of H, i.e., the graph $G = (V(G), E(G))$, with $V(G) = V(H)$, such that $E(G)$ contains an edge between two vertices u, v if and only if u, v jointly occur in an edge in $E(H)$. Clearly, every edge of a hypergraph H gives rise to a clique in the Gaifman graph G. Moreover, it is easy to verify that every clique of a graph has to be contained in some bag of a TD. Hence, for TDs, the Gaifman graph contains all the relevant information of the

hypergraph. In contrast, for HDs and GHDs, we additionally have to keep track of the edges of the hypergraph, since only these are allowed to be used in the λ-labels. We have, therefore, preferred to define also TDs directly for hypergraphs (rather than via the Gaifman graph).

CHAPTER 3

Related Work

In this chapter we will cover related topics most relevant for our work. We start by covering related work in acyclic and cyclic query answering, which is of central importance in this thesis.

Acyclic Queries The algorithm for evaluating acyclic queries, presented by Yannakakis over 40 years ago [162], has long been central to the *theory* of query processing. In recent years, this approach to query evaluation has gained renewed momentum in *practice* as evidenced by several extensions and applications. Recent work has aimed at bringing its advantages into DBMSs from the outside via SQL query rewriting [38, 154], and similar methods such as generating Scala code expressing Yannakakis' algorithm as Spark RDD-operations [43].

Multiple recent works [84, 85, 155], propose extensions of Yannakakis' algorithm for dynamic query evaluation. Further research extends and applies Yannakakis' algorithm to comparisons spanning several relations [156], queries with theta-joins [85], differences of CQs [81], and privacy preserving query processing [157]. An important feature of Yannakakis' algorithm is the elimination of *dangling tuples* (i.e., tuples that do not contribute to the final result) via semi-joins. In a recent paper [83], a new join method was introduced that integrates the detection and elimination of dangling tuples into the join computation. Very recently, there has been an explosion of interest on the integration of Yannakakis-style query evaluation into DBMSs while avoiding the overhead of *several* traversals of the join tree via semi-joins and joins. Birler et al. [23] have coined the term *diamond problem* to describe large intermediate results in join queries, and integrated a Yannakakis-like approach into the compiled query engine of Umbra [121] by splitting the join operation into "lookup" and "expand" operators. Bekkers et al. [21] perform a similar integration into the interpreted query engine Apache DataFusion [96]. A related approach – *Predicate Transfer* [161] – and specifically *Robust Predicate Transfer*, introduced by Zhao et al. [169], was integrated into DuckDB.

Decompositions An important line of research has extended the applicability of Yannakakis-style query evaluation to "almost acyclic" queries. Here, "almost acyclic" is formalized through various notions of decompositions such as (normal, generalized, or fractional) hypertree decompositions [63, 6, 74]. Each of these decompositions is associated with a notion of "width" that measures the distance from acyclicity, with acyclic queries having a width of 1. Several works [2, 43, 130, 152], combine Yannakakis-style query evaluation based on various types of decompositions with multiway joins and worst-case optimal join techniques.

Aggregate Queries Aggregates are commonly used on top of join queries – especially in data analytics. Green et al. [71] gave a new perspective on aggregate queries by considering K-relations, i.e., relations annotated with values from some semi-ring K. Join queries over K-relations then come down to evaluating sum-product expressions over the underlying semi-ring. The combination of aggregate queries with Yannakakis-style query evaluation was studied in the FAQ-framework (Functional Aggregate Queries) [91] and, similarly, under the name AJAR (Aggregations and Joins over Annotated Relations) [90]. A crucial problem studied in both papers is the interplay between the ordering of a sequence of aggregate functions and (generalized or fractional) hypertree decompositions. In both papers, the ultimate goal is an efficient, Yannankakis-style evaluation algorithm for aggregate queries based on finding a good variable order. Similar ideas to FAQs and AJAR queries also appear in earlier works on joins and aggregates over factorized databases [14, 127] and on quantified conjunctive queries (QCQs) [39]. A general framework for hybrid database and linear algebra workloads (as are typical for machine learning applications) has recently been proposed by Shaikhha et al. [142]. It provides a performant, unified framework for data science pipelines by introducing the purely functional language SDQL and combining optimisation techniques from databases (e.g., pushing aggregates past joins) and linear algebra (e.g., matrix chain ordering).

Homomorphism Counting From a logical perspective, evaluating join queries, finding homomorphisms, and solving CSPs are equivalent problems [72]. Hence, the problem of counting homomorphisms [44] (in particular, counting graph homomorphisms) can be seen as a special case of aggregates on top of join queries, which has received a lot of research interest (see, e.g., [45, 115, 26]. In [166], the problem of counting graph homomorphisms is indeed treated as a query evaluation problem for aggregates on top of joins. A key idea here is to push group-by and aggregation over joins in a generalized hypertree decomposition.

Constant Delay Enumeration Yannakakis' algorithm has received a lot of attention in the Database Theory community in the context of identifying classes of queries that allow for particularly efficient enumeration of query result, namely linear-time pre-processing and constant delay. This line of research was initiated by Bagan et al. [13] and has triggered a lot of follow-up work such as [33, 34, 35, 36, 55, 106] since then.

Reducing the Number of Join Computations Several works have addressed the need to compute a high number of joins in different contexts and have aimed at reducing this number. The work by Schleich et al. [139] on LMFAO (Layered Multiple Functional Aggregate optimisation) specifically targets machine learning applications that require the computation of large batches of aggregate queries over the same join query. A dramatic speed-up is achieved by decomposing aggregates into views and arranging them at nodes in a join tree to avoid the re-computation of the same intermediate joins time and again. In principle, the need to re-compute similar joins time and again also arises in the area of IVM (incremental view maintenance). A revolutionary approach to IVM was proposed by Koch et al. [94] with the DBToaster system, that avoids the re-computation of joins in case of updates to the database by maintaining "higher-order" delta views, i.e., delta queries (= first-order deltas), delta queries to the deltas (= second-order deltas), etc.. A further performance gain is achieved with F-IVM (factorised IVM) [125], that groups various aggregates together and thus reduces the number of views to be maintained. Moreover, factorization is applied, for instance, to avoid the materialisation of Cartesian products in views. Of course, independently of IVM, factorization [126] is a generally applicable method to keep the query result in a compressed form and avoid its complete materialisation.

Distributed ACQ Processing The potential of applying Yannakakis-style query evaluation to distributed processing comes from the fact that the evaluation of ACQs lies in the highly parallelizable class LogCFL [61]. This favorable property was later extended to "almost acyclic" queries by establishing the LogCFL-membership also for queries with bounded hypertree width [63]. A realization of Yannakakis' algorithm in MapReduce [7] further emphasized the parallelizability of Yannakakis-style query evaluation.

Spark and Spark SQL Spark, a top-level Apache project since 2014, is often regarded as a further development of the MapReduce processing model. Spark SQL [11] provides relational query capability within the Spark framework. Query optimisation is a primary focus of Spark SQL, with the powerful Catalyst optimiser being an integral component since its inception [11]. Several later works [143, 165, 86, 15, 118]) have proposed further measures to speed up query processing in Spark SQL. The recently presented SparkSQL$^+$ system [43] combines decompositions and worst-case optimal join techniques as well as the optimisations for CQs with comparisons spanning several relations [156] and allows users to experiment with different query plans. Zhang et al. [166] recently implemented specific worst-case optimal join algorithms in combination with decomposition-based methods on top of Spark SQL as part of a system focused specifically on subgraph counting.

Query Rewriting Optimising queries before they enter the DBMS is a different strategy towards query optimisation that has been successfully applied in standard DBMSs [171, 172]. Although DBMSs already perform optimisations on the execution of the query, it has been shown that rewriting the query itself can still be highly effective.

The WeTune [158] system goes even further, and can be used to automatically discover rewrite rules but comes with the disadvantage of extremely long runtimes.

Machine Learning for Databases There has been growing interest in the application of machine learning techniques to increase the performance of database systems, as can be seen by a recent survey on this broad area [170]. We proceed to give a very brief overview of the general topics as to how machine learning has been adapted for database research. For a more detailed account on the rich interaction between machine learning and databases, we refer to [170]. In this survey, the authors categorize the different efforts of using machine learning for core database tasks into several groups. The first group is "learning-based data configuration". These are works that aim to utilize machine learning for knob tuning, and view advisor and index advisor tasks [48, 167, 175, 9, 104, 148, 79, 124, 37, 140]. Related work that also falls into this category is presented in [111, 112]. The next group is "learning-based data optimisation". These works aim to tackle important, computationally intractable problems such as join-order selection and cardinality estimation of joins [119, 129, 128, 49, 52, 93, 113, 114, 78, 145, 151, 153]. Another group is "learning-based design for databases". These works aim more specifically at exploring the use of machine-learning in the construction of various data structures used by modern databases, such as indexes, hashmaps, bloom filters and so on [95, 47, 160, 107]. A further group listed in the survey is "learning-based data monitoring". As the name suggests, these works aim to use machine learning to create systems that automate the task of running a database and detecting and reacting to anomalies [108, 147, 75, 114, 173]. Lastly the survey mentions "learning-based database security". This category is on how to use ML methods to help with critical problems, such as confidentiality, data integrity and availability [22, 42, 105, 144, 19].

Query Performance Prediction Predicting the performance of a query – usually the runtime, or sometimes other resource requirements – is related to the problem of deciding whether to rewrite a query. Runtime prediction has been performed by constructing cost models based on statistical information of the data [77], on SQL queries [159], and XML queries [173]. Further approaches use machine learning and deep learning to predict the runtimes of single queries [168, 114, 174] or concurrent queries (workload performance prediction) [50, 8].

CHAPTER 4

Integrating Yannakakis' Algorithm into Database Systems

In this first main chapter, we begin by exploring the integration of Yannakakis' algorithm into systems from the outside via query-rewriting. After showing the feasibility of this approach and identifying weaknesses, we will proceed to make it practicable. To achieve this, we will consider the problem of applying the rewriting as an algorithm selection problem, and subsequently develop methods for solving this problem.

In Section 4.1, we introduce the class of *zero-materialisation answerable* queries (0MA queries, for short), which can be evaluated by semi-joins only. The idea of our rewriting-based approach for combining structure-guided query evaluation with traditional DBMS technology and the experimental results thus obtained are presented in Section 4.3. In Section 4.4, we describe the algorithm selection problem, and we present a methodology for solving this problem in Section 4.5. Empirical results are presented in Section 4.6. We summarise the results of this Chapter in Section 4.7.

4.1 0MA Queries

It is well known [61] that for Boolean ACQs (i.e., queries where we are only interested in whether the answer is non-empty), Yannakakis' algorithm can be stopped after the first bottom-up traversal. Indeed, if at that stage the relation associated with the root node of the join tree is non-empty, then so is the query result. Most importantly, for queries of this type, the most expensive part of the evaluation (i.e., the joins in the second bottom-up traversal) can be completely omitted. The next example illustrates that such favourable behaviour is by no means restricted to Boolean queries.

Example 4.1.1. *Consider an excerpt of a university schema with relations* exams(cid, student, grade) *and* courses(cid, faculty)*. Querying each student's lowest grade in courses of the Biology faculty is naturally stated in SQL as follows.*

```
SELECT   exams.student, MIN(exams.grade)
FROM     exams,courses
WHERE    exams.cid=courses.cid
         AND courses.faculty='Biology'
GROUP BY exams.student;
```

Ignoring the GROUP BY clause for a while, the query involving only two relations is trivially acyclic. In the join tree consisting of 2 nodes, we choose as root the node labelled by the exams*-relation. After the first bottom-up traversal, this relation contains all* exams*-tuples that join with the* courses*-relation restricted to those tuples with faculty = 'Biology'. Hence, if we now also take the GROUP BY clause into account, answering the query is possible by only looking at the* exams*-relation – without the need for the remaining two traversals of Yannakakis' algorithm.* ◇

In this section, we want to identify a whole family of queries whose evaluation only requires the first bottom-up traversal of Yannakakis' algorithm. To this end, we introduce the class of *zero-materialisation answerable (0MA)* queries and we will illustrate the usefulness of this class by various examples. In particular, Boolean queries and the query from Example 4.1.1 are contained in this class. The performance gain attainable when answering 0MA queries will be demonstrated experimentally in Section 4.3.

Similar to the syntax for aggregation and grouping in the Extended Relational Algebra as introduced in Chapter 2, where we would write $\gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)]$ to denote the aggregation/group by operator, we write here, with subscript notation for better readability: $\gamma_{g_1,\ldots,g_\ell,\ A_1(a_1),\ldots,A_m(a_m)}$.

Definition 4.1.1.

- *A query Q is in* aggregation normal form[1] *if it is of the form $\gamma_U(\pi_S(Q'))$, where Q' is a query consisting only of natural joins and selection.*
- *For a query Q in aggregation normal form, we say that Q is* guarded*, if Q' mentions a relation R with $Att(S) \subseteq Att(R)$, i.e., R contains all attributes occurring in the GROUP BY clause (aggregate or not). If this is the case, we say that R* guards *query Q or, equivalently, R is a* guard *of Q.*
- *We say that a query $Q = \gamma_U(\pi_S(Q'))$ is* set-safe *if it is equivalent to $\gamma_U(\delta(\pi_S(Q')))$, i.e., duplicate elimination before the GROUP BY does not change the meaning of the query.*

[1] Note that the γ operator also implicitly projects to some subset of attributes. The projection π_S is thus not strictly necessary and is only added for clarity.

- *A query Q in aggregation normal form is called* zero-materialisation answerable *(*0MA*) if it is guarded and set-safe.*

As far as the notation is concerned, recall from Chapter 2 that the restriction to natural joins and top-level projection is without loss of generality and it only serves to simplify the notation. This is also the case in the above definition. In our examples, we may freely lift this restriction if it is convenient. In contrast to the restricted notation of CQs in Chapter 2, we now prefer to make selection explicit in the "inner" query Q' – in addition to the natural joins.

Despite the various technical constraints, 0MA queries still cover many common query patterns. Clearly, the restriction to aggregation normal form matches the standard use of aggregates in SELECT-FROM-WHERE-GROUP BY statements in SQL. Also the further restrictions imposed by 0MA queries are met by many common query patterns observed in practice. Boolean ACQs mentioned above (e.g., realised by a query of the form SELECT 1 FROM ....) are a special case of 0MA queries, where we simply leave out the grouping, and the projection is to the empty set of attributes.

We next verify that also the query from Example 4.1.1 is 0MA. By slightly simplifying the subscripts (in particular, abbreviating attribute names), the query translates to the following Relational Algebra query Q:

$$\gamma_{\text{stud,MIN(grad)}}(\pi_{\text{stud,grad}}(\mathsf{exams} \bowtie \sigma_{\text{faculty}=\texttt{'Biology'}}(\mathsf{courses})))$$

Clearly, query Q is zero-materialisation answerable, since relation exams (containing both attributes student and grade) is a guard of Q and aggregation via MIN (or MAX) is always set-safe.

We now formally prove that acyclic 0MA queries may indeed be evaluated without the join-phase of Yannakakis' algorithm. That is, these queries can be evaluated via aggregate/group processing over a single relation of the database that has been reduced by the semi-joins of the first bottom-up traversal.

Theorem 4.1.2. *Let $Q = \gamma_U(\pi_S(Q'))$ be a 0MA query in aggregation normal form such that Q' is an ACQ, and let D be an arbitrary database. Let $\langle T, r, \lambda\rangle$ be a join tree of Q' such that the root r of T is labelled by relation R that guards Q. Let R' be the relation associated with node r after the first bottom-up traversal of Yannakakis' algorithm. Then the equality $Q(D) = \gamma_U(\delta(\pi_S(R')))$ holds.*

Proof Sketch. After the first bottom-up traversal, all tuples in a relation associated with a node in T actually join with all relations in the subtree below. Since R' is the relation at the root, every tuple $r \in R'$ extends to a result of Q'. Since Q is guarded, we have that S is a subset of attributes in R' and thus $\pi_S(Q'(D)) \supseteq \delta(\pi_S(R'))$ and, therefore, also $\delta(\pi_S(Q'(D))) \supseteq \delta(\pi_S(R'))$.

On the other hand, since R is part of Q', which consists only of natural joins and selection, any tuple in $Q'(D)$ must be consistent with R. Since every tuple in $Q'(D)$ must also be

consistent with all other relations mentioned in Q', it must also be consistent with R' and, therefore, $\delta(\pi_S(Q'(D))) \subseteq \delta(\pi_S(R'))$ holds. Moreover, as Q is set-safe, we also have $\gamma_U(\pi_S(Q'(D))) = \gamma_U(\delta(\pi_S(Q'(D))))$ and, hence, $\gamma_U(\delta(\pi_S(Q'(D)))) = \gamma_U(\delta(\pi_S(R')))$. □

Note that the requirement in Theorem 4.1.2 that the guard R must be the label of the root node of a join tree of Q' does not impose any additional restrictions apart from the conditions that R must be a guard and Q' must be an ACQ. Some node in the join is guaranteed to be labelled by R, and we can always choose this particular node as the root of the join tree.

It is important to note that set-safety is not required due to any technical issues with bag semantics. The restriction to set-safety is only needed to identify queries whose answer can be determined without knowing the exact multiplicity of a tuple in the answer. As far as standard aggregate functions are concerned, this always holds for MAX and MIN as we have seen in Example 4.1.1. In contrast, other standard aggregates, such as SUM or COUNT are, in general, not set-safe. They nevertheless can be answered efficiently when knowing the multiplicity of each tuple in the result of the join query Q' (for further discussion on this, refer to Chapter 5). These aggregates may actually be used in patterns that are set-safe, e.g., COUNT(DISTINCT ...) constructs in SQL. Indeed, it is easy to see that the combination with DISTINCT can make any aggregate set-safe. Furthermore, trivial use of γ as projection (all attributes are grouping attributes) also covers the enumeration of distinct tuples as a set-safe operation. In practice, even more cases may be set-safe due to constraints on the data such as, for instance, counting the different values of an attribute with a UNIQUE constraint.

Deciding the 0MA Property. It is natural to consider the question of deciding whether a given query is 0MA. We give a brief informal discussion of why this is not of particular interest in our case. First, it is clear that deciding whether a query is guarded is trivial, and only deciding the set-safe condition is of any real concern. For the inner query Q', the set-safety status boils down to the question if it can return duplicate results or not, which is well understood and easy to check (recall that we are restricting ourselves to CQs with some extensions such as GROUP BY, HAVING, aggregates; so undecidability results for FO queries such as non-emptiness do not apply here): if multiset input relations are allowed, then every query may possibly return duplicate tuples; otherwise any query where some attribute is projected out can return duplicates.

Consequently, only the semantics of the aggregate functions themselves are the important factor for set-safety. In general, set-safety is a non-trivial property of the aggregate functions and thus expected to be undecidable if we allow *arbitrary* computable functions as aggregates. However, we are interested in the concrete behaviour of current DBMSs, which typically only offer a small fixed vocabulary of aggregation functions. For instance, the ANSI SQL standard specifies 28 possible aggregation functions and they are easy

to check for set-safety case by case, without the need for a general procedure to check set-safety of arbitrary functions.

4.1.1 More General Queries

In this section, we inspect several situations in which we are not dealing with acyclic CQs and/or not zero-materialisation answerable queries, and where the performance gain achieved by a short-cut in Yannakakis' algorithm is nevertheless attainable.

Recall that we have omitted a HAVING clause from our aggregation normal form in Definition 4.1.1. If we have a 0MA query with a HAVING clause on top of it (see, e.g., Example 4.1.4 below), then we can still evaluate the 0MA query without materialising any joins and simply filter the result by the HAVING condition afterwards.

More generally, the optimisation from Theorem 4.1.2 is applicable whenever some part of a query satisfies the 0MA condition. For instance, subqueries with the EXISTS operator are actually Boolean queries and, as such, 0MA – provided that they are ACQs.

The following example involving a 0MA subquery is taken from the TPC-H benchmark:

Example 4.1.3. *TPC-H Query 2 contains the following subquery:*

```
SELECT MIN(ps_supplycost)
FROM partsupp, supplier, nation, region
WHERE p_partkey = ps_partkey AND ...
```

where `p_partkey` *is an attribute coming from the outer query and the rest of the WHERE clause are equi-joins and selections. This subquery is a standard example of a 0MA query, since aggregation by MIN is always set-safe and the query is clearly guarded by* `partsupp`*. The subquery is correlated inside the TPC-H Query 2 due to the attribute* `p_partkey` *from the outer query, but it allows for effective decorrelation. Notably, if we consider magic decorrelation [141], then we would change the select clause of the subquery to* `ps_partkey, min(ps_supplycost)`*, add a grouping over* `ps_partkey`*, and remove the correlated join with* `p_partkey`*. This transformation preserves guardedness and set-safety and we could, in this case, combine decorrelation with the efficient evaluation of the decorrelated 0MA subquery according to Theorem 4.1.2.* ◇

Below we see a more complex TPC-H query, where optimised evaluation based on 0MA-parts is even possible twice – once for the subquery and once for the outer query.

Example 4.1.4. *TPC-H Query 11 is of the following form*

```
SELECT ps_partkey,
       SUM(ps_supplycost*ps_availqty)
FROM partsupp, supplier, nation
```

```
WHERE ps_suppkey = s_suppkey
  AND s_nationkey = n_nationkey
  AND n_name = 'GERMANY'
GROUP BY ps_partkey
HAVING SUM(ps_supplycost*ps_availqty) >
  (SELECT SUM(ps_supplycost*ps_availqty)
              * 0.0001
   FROM ...)
```

where the omitted FROM clause of the subquery is the same as the FROM clause of the outer query. That is, the subquery is almost the same as the outer query: we just leave out the grouping by `ps_partkey`*, and the sum over* `ps_supplycost` $*$ `ps_availqty` *is now taken over all* `ps_partkey`*'s and is multiplied by 0.0001.*

At its core, this SQL query can be evaluated via a 0MA query of the form $Q = \gamma_U(\pi_S(Q'))$*, where* Q' *represents the join query on the three relations, and*

$$U = \texttt{ps\_partkey}, \texttt{ps\_suppkey}, \texttt{ps\_supplycost} * \texttt{ps\_availqty}.$$

Note that keeping `ps_suppkey` *in the grouping at this step is important to observe that the essence of this query is set-safe. The result of both, the outer query and the subquery in the HAVING clause, can be directly obtained from* Q*, leaving only a final filtering step.*

We now analyse why Q *is 0MA. While* `partsupp` *clearly guards the query, observing set-safety requires a small but natural step beyond the technical definition above. In TPC-H, there are constraints on the database that require that* `s_suppkey` *and* `n_nationkey` *be keys of* `supplier` *and* `nation`*, respectively. Therefore, every tuple in* `partsupp` *can have only one join partner in* `supplier`*, and the result has only one join partner in* `nation`*. Furthermore, the projection on* Q' *retains the key (*`ps_partkey`*,* `ps_suppkey`*) letting us observe overall that every tuple in the result of* Q' *is in fact distinct and, as a consequence,* Q *is also set-safe.* ◇

We conclude this section by briefly discussing CQs to which Theorem 4.1.2 is not applicable. That is, either acyclicity or the 0MA property is violated. In case of cyclic queries, we may apply decomposition methods [63, 74] to turn a given CQ into an acyclic one. Since CQs in practice tend to be acyclic or almost acyclic [25, 54], this transformation into an ACQ is in theory feasible at the expense of a polynomial blow-up (where the degree of the polynomial is bounded by the corresponding notion of width). Actually, the queries from the benchmark of [110], which we use for our experimental evaluation, follow this pattern: the vast majority of the queries are acyclic and the rest have low generalised hypertree width (ghw). We will present first experiments with queries of low ghw (see Section 4.3.4) and later develop more sophisticated techniques for cyclic queries in Chapter 6. We will cover queries where the 0MA property is violated in Chapter 5, where we will also study the prevalence of 0MA queries and more general classes in standard benchmarks.

4.2 Realising Yannakakis' Algorithm via Query Rewriting

Our goal is to shed light on the benefits of realising structure-guided query evaluation for common database systems. We thus do not want to restrict ourselves to a single system nor to a single architecture or a single query planning and execution strategy. Therefore, we have chosen three DBMSs based on different technologies: PostgreSQL 13.4 [146] as a "classical" row-oriented relational DBMS, DuckDB 0.4 [134] as a column-oriented, embedded database, and Spark SQL 3.3 [164] as a database engine specifically designed for distributed data processing in a cluster.

We have implemented a proof-of-concept system, referred to as YanRe, that works by rewriting a query into a sequence of SQL statements, which express Yannakakis' algorithm. This makes our approach easily portable and we can apply it to the three chosen DBMSs with almost no change to our rewriting method (apart from some minor differences in SQL syntax). The significant effort of a full integration into any of the three systems (let alone, into all of them) does not seem to be justified before gathering further information on the potential benefit of such an integration. Moreover, our rewriting-based approach is also applicable to commercial DBMSs, where large internal modifications are impossible, or situations where modifications are not feasible due to dependencies on specific versions or hosting in "the cloud". In our experiments, we compare the performance of join queries in each DBMS with the performance of the YanRe rewriting, executed by the same system.

The YanRe system proceeds in several steps: we first extract the CQ from the given SQL query and transform it into a hypergraph. From this, we compute a join tree by applying a variant of the GYO-algorithm [68, 163] described in the next section. We then generate the SQL statements that correspond to the semi-joins and joins of Yannakakis' algorithm. These SQL statements involve the creation of temporary tables. If the original query contains GROUP BY and HAVING clauses or more general selections (beyond equalities), then these can be integrated into the SQL statement referring to the root node in the final traversal of the join tree.

The queries in the benchmark of [110] are all straightforward SELECT-PROJECT-JOIN queries (in particular, no GROUP BY and HAVING clauses, no subqueries). We process these queries via a simplified version of the SQL-to-CQ translation from [54], which also provides the further translation of the CQ into a hypergraph. Recall that the hypergraph $H = (V, E)$ of a CQ Q is obtained by identifying the vertices in V with the variables in Q and defining as edges in E those sets of vertices where the corresponding variables occur jointly in an atom of Q. The join tree computation and the generation of SQL statements are discussed below in more detail.

4.2.1 Join Tree Computation

The GYO algorithm [68, 163] for deciding whether a hypergraph (and thus the corresponding query) is acyclic works by non-deterministic application of the following steps:

Algorithm 4.1: The Flat-GYO algorithm

```
   input  : A connected α-acyclic hypergraph H
   output : A join tree of H
 1 J ← empty tree;
 2 while H contains more than 1 edge do
 3 |   Delete all degree 1 vertices from H;
 4 |   for e ∈ E(H) s.t. there is no f ∈ E(H) with e ⊂ f do
 5 |   |   C_e ← {c ∈ E(H) | c ⊆ e};
 6 |   |   for c ∈ C_e do
 7 |   |   |   Set label(c) as child of label(e) in J;
 8 |   |   |   Remove c from H;
 9 |   |   end
10 |   end
11 end
12 return J;
```

i) deleting a vertex with degree 1 (i.e., a vertex occurring in a single edge), ii) deleting an empty edge, or iii) deleting an edge that is a subset of another edge. In Algorithm 4.1, we choose a particular order in which the elimination steps of the GYO-algorithm are executed. Technically, deletion of degree 1 vertices from an edge e of H may produce a new edge that is not part of the join tree. We thus use label(e) in Algorithm 4.1 to always refer to the name of the original edge before vertex removals. The algorithm produces join trees with a particular property expressed in the following theorem:

Theorem 4.2.1. *Let $H = (V(H), E(H))$ be an acyclic hypergraph and let T denote the join tree resulting from applying Algorithm 4.1 to H. Then T has minimal depth among all join trees of H.*

Proof. The proof proceeds in three steps: (1) First, we observe that there is still some non-determinism left in Algorithm 4.1, that depends on the order in which the edges in the for-loop on line 4 are processed. It may happen (i) that $e = e'$ holds for two edges with label(e) $\neq$ label(e') and that (ii) for two distinct maximal edges e, e', an edge $c \in E(H)$ satisfies both $c \subseteq e$ and $c \subseteq e'$ on line 5. Nevertheless, the number of iterations of the while-loop is independent of the order in which the maximal edges are processed in the for-loop. This property follows from the easily verifiable fact that the set of edges $\{e_{i_1}, \ldots, e_{i_m}\}$ resulting from an iteration of the while-loop is independent of this non-determinism, even though (due to (i)) there may be an alternative set of edges with different labels and (due to (ii)) also an alternative collection of parent/child relationships may be possible.

(2) Second, if a run of Algorithm 4.1 has k iterations of the while-loop, then the join tree constructed by this run has at most depth k (max. distance from root to leaf). This is due to the fact that, on line 7, existing partially constructed trees may be appended

below a new root node but no further nesting may happen here. Hence, the depth of the partially constructed trees grows by at most 1.

(3) Finally, if there exists a join tree T of depth k, then there exists a run of Algorithm 4.1 with at most k iterations of the while loop. This property is proved by a simple induction argument: there exists an order in which the maximal edges are processed in the for-loop, so that all leaf nodes of T get removed on line 8 – thus decreasing the depth of T by at least 1.

The theorem can then be proved as follows: Suppose that, for a given hypergraph H, the minimum depth of any join tree of H is k. Then there exists a join tree T of depth k. Hence, by (3), Algorithm 4.1 has a run with at most k iterations of the while-loop and, therefore, by (1), any run of Algorithm 4.1 has a run with at most k iterations of the while-loop. Thus, by (2), any run of Algorithm 4.1 produces a join tree of depth at most k. □

4.2.2 Query Plan Generation and Execution

In the next step, we create a sequence of SQL statements that express the execution of Yannakakis' algorithm over the join tree and reintroduce final projection and aggregation if applicable. The overall evaluation of the query is thus split into four stages, which we briefly describe below. We will illustrate these steps by means of the SQL query given in the following example.

Example 4.2.2. *Recall the university schema of Example 4.1.1 with relations* exams(cid, student, grade) *and* courses(cid, faculty). *We now add the two relations* tutors(student, cid, num_semesters) *and* enrolled(student, program). *The following query retrieves, for each fixed pair of program and course, the lowest grade obtained in exams of the CS faculty by any student enrolled in that program and who has been tutored for more than 1 semester in that course.*

```
SELECT enrolled.program, exams.cid,
       MIN(exams.grade)
FROM exams, courses, enrolled, tutors
WHERE exams.cid = courses.cid
  AND exams.student = enrolled.student
  AND exams.cid = tutors.cid
  AND courses.faculty = 'ComputerScience'
  AND exams.student = tutors.student
  AND tutors.num_semesters > 1
GROUP BY enrolled.program, exams.cid;
```

The query is acyclic but not 0MA (it is not guarded). Its hypergraph and a possible join tree are depicted in Figure 4.1, where, for the sake of readability, the names of vertices are abbreviated to the first character. ◇

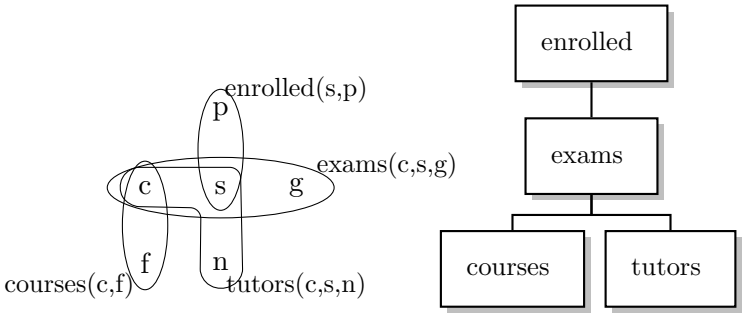


Figure 4.1: Hypergraph and join tree for Example 4.2.2

The Setup Stage We first rename the attributes in such a way that all equi-joins are replaced by natural joins throughout the rest of the process. Then, from the join tree perspective, we create one view per node, representing the relation in the join tree before the execution of Yannakakis' algorithm. Early projection to the attributes which are actually used in the query (either as a join attribute or as part of the final result) as well as applicable selections are also incorporated directly into these views. For instance, for the query and join tree from Example 4.2.2, the leaf node for relation courses induces the following view `courses_setup`:

```
CREATE VIEW courses_setup AS SELECT cid
FROM courses WHERE faculty='ComputerScience';
```

The Semi-Join Stages The views from the setup stage are used to generate SQL statements for the semi-joins of the first bottom-up traversal and, if the query does not satisfy the 0MA-property, also for the top-down traversal of the join tree. The result of each semi-join is stored in an auxiliary temporary table. Semi-joins are expressed in the standard manner via the IN operator of SQL.

To illustrate the semi-join stages, we continue our example from above. Assuming that all views from the setup stage are named with the `_setup` suffix, the first semi-joins of the bottom-up traversal are realised in SQL as follows (for clarity, the previously mentioned renaming of attributes is not performed here):

```
CREATE TEMP TABLE exams_sjup AS
SELECT * FROM exams_setup WHERE
cid IN (SELECT cid from courses_setup) AND
cid, student IN (SELECT cid, student
                 FROM tutors_setup);
```

We thus create a new intermediate relation for the exams node. Importantly, the analogous statement expressing the semi-join from the `exams` node into the `enrolled` node will now make use of `exams_sjup` rather than the setup view for the `exams` node.

The Join Stage Finally, the temporary tables representing the relations after the semi-join stages are combined by natural joins. The straightforward way to do this is

either via step-wise joins along the join tree in a bottom-up manner or, alternatively, all relations can be joined in one large statement. The latter option seems to introduce less overhead, but for large original queries, it reintroduces the problem of planning queries with many joins. We therefore take a middle ground and group (via a straightforward greedy procedure) the join tree into subtrees of at most 12 nodes each and materialise the final joins with one join query per subtree, plus a final query joining the subtrees. Of course, for 0MA queries, no computation of joins is necessary. In this case, the join phase simply refers to the final aggregation over the root node.

Finally, note that these stages are also amenable to parallelisation: as we follow a tree structure, we know that the semi-joins and joins for nodes in different subtrees can be computed independently of each other. This thread is not further followed in this work as the host systems considered here already parallelise query execution to an extent where further parallelisation "from the outside" does not seem particularly helpful. However, the additional potential of parallelisation clearly deserves further study.

4.3 Performance Evaluation of Yannakakis' Algorithm

In this section, we detail the results of our experiments, which demonstrate that structure-guided query evaluation can indeed greatly improve performance on challenging join queries.

4.3.1 Experimental Setup

We perform experiments using a recent benchmark by Mancini et al. [110], which consists of 435 challenging synthetic join queries over the MusicBrainz dataset [1]. We do not yet focus on classic benchmark datasets, such as from TPC-H or TPC-DS, later, as they are less interesting for our purposes since their focus is not on the complexity of evaluating queries with a large number of potentially expensive joins. In contrast, the benchmark from [110] that we consider here contains queries with as many as 30 relations and, in many cases, the join processing (as well as planning, see [110]) is very challenging for modern DBMSs. The queries in this benchmark were created over the MusicBrainz dataset [1] by randomly joining tables along foreign key relationships. This makes the generated queries similar to real-world queries and particularly interesting for our experiments since one would normally expect classical DBMSs to perform particularly well on this kind of queries. Moreover, the large number of generated queries protects against the very significant variance in the evaluation of large queries. Another important reason for choosing the queries from this benchmark is that they operate on a *publicly available* dataset, which makes our results fully reproducible. This is in sharp contrast to big join queries mentioned in other works such as [46, 122].

We will report on two types of experiments. One set of tests will be referred to as *full enumeration* queries. For these, we essentially use the original queries of [110]. However, since these queries contain no projection, we adapt the queries to project to only the join

attributes (one attribute per equi-join equivalence class, i.e., no redundant columns) in order to lessen the role of unimportant I/O. In a second set of experiments, we explore the effectiveness of computing aggregate queries with the 0MA property from Definition 4.1.1. For this purpose, we transform each query to compute a "MIN" aggregate for an attribute that we randomly choose from those attributes that already occur in the original query. In the following, we refer to these aggregation variants as the *0MA aggregation* queries. In both cases, the queries are always executed on the standard MusicBrainz dataset. For all experiments in this section, we use a timeout of 20 minutes for the execution of each query. The experiments on DuckDB and PostgreSQL are performed on a machine with an Intel Xeon Bronze 3104 with 6 cores clocked at 1.7 GHz, and 128 GB of RAM and running Debian 11, using the Linux kernel 5.10.0 with all data stored on an SSD. The default settings of PostgreSQL proved unsatisfactory in our system environment. We therefore explicitly configured PostgreSQL to use at most 8 concurrent working threads and 200 concurrent I/O requests, which turned out to be the most suitable configuration for our system. For DuckDB, we use all default parameters (leading to full utilisation of all cores and concurrent disk I/O). Our experiments with Spark SQL are performed in a cluster environment with two namenodes and 18 datanodes, with each node having two XeonE5-2650v4 CPUs with 24 cores (48 per node) and 256 GB RAM.

In addition to reporting the results in this section, we also provide all raw data of our experiments and instructions for reproducing them on Figshare `https://figshare.com/s/b9ba4b79876ocf6af3a4`. We include there only the rewritten queries, as were produced by YanRe and detailed logs of their execution. We omit the original queries from [110] and we hope to make the full data publicly available in the future.

4.3.2 Experimental Results

We primarily concentrate on acyclic queries from the benchmark of [110]. ACQs form the majority of the benchmark, namely 351 out of 435. The number of relations in the queries of this benchmark lies between 2 and 30. Further details on cyclic queries are given in Section 4.3.4.

Table 4.1 summarises our results for the ACQs in the benchmark. The Mean, Med. (Median), and Std. Dev. columns report statistical information for the running times of the benchmark queries. Queries timed out (i.e., which did not terminate within 20 minutes), are counted as having running time 20 minutes. The Max column reports the maximum running time of the queries that did not time out. The number of queries that did not terminate within the time limit is stated in the Timeouts column. Recall, that the Spark SQL experiments were performed on a significantly different environment and our experiments are not intended or suited for direct comparison of times between different baseline systems.

We see that the number of queries that execute within the time limit of 20 minutes is significantly higher when using YanRe: in DuckDB, the use of YanRe reduces the number of timeouts from 69 to 29 for full enumeration queries, and from 58 to no timeouts

at all for 0MA aggregation queries. In Spark SQL, this reduction is from 87 to 29 and from 91 to 3, respectively. Consequently, we also see an improvement of up to factor 2 in the mean running times[2]. In PostgreSQL, we see a reduction from 97 to 70 timeouts in the case of full enumeration queries, and from 91 timeouts to just 2 in the 0MA aggregation case. Furthermore, those additional queries that terminate within the time limit do so with a very clear margin as can be seen from the maximum times.

The low median, contrasting the much higher means, shows that at least half of the queries are reasonably easy to solve for the baseline systems. This is expected, as the number of relations is uniformly distributed in the queries, meaning that a fair amount of queries are small enough for typical query planning strategies to work well. This observation is discussed in further detail below. The split into multiple SQL statements as well as the creation of various temporary tables as performed by YanRe naturally leads to some overhead. This is clearly visible in the higher median execution time with YanRe. This comes as little surprise, since the structure-guided approach is most effective for hard cases.

Beyond the general improvement, we observe a particularly large improvement for 0MA queries. For DuckDB + YanRe, not only are all queries solved within the 20 minute time limit, but all are solved within only 16 *seconds*. In case of Spark SQL + YanRe and PostgreSQL + YanRe, even though the mean speed-up is smaller, we still observe an improvement by an order of magnitude as well as the elimination of almost all timeout cases (see below for further discussion of the remaining timeouts). While we performed minimum aggregation in our experiments, any natural 0MA version of the queries (e.g., counting or enumerating distinct values of some attribute) would result in essentially the same running times in the YanRe cases. Note however, that the median times are still slightly lower without YanRe, again demonstrating that the structure-guided approach is particularly well suited for complementing traditional query processing strategies in difficult cases. Ultimately, we see that the use of YanRe makes these types of queries feasible, whereas our experiments show that none of the baseline systems tested here can be relied upon to produce answers for such queries within a reasonable time limit.

While our results for all three systems follow the same trends, we see that PostgreSQL performs significantly worse. In particular, the 2 timeouts for 0MA aggregation queries with YanRe on top of PostgreSQL are surprising and merit discussion. In these two cases, the join trees contain nodes with a large number of children. The resulting SQL statement generated by YanRe therefore expresses many semi-joins at once. While this is not an issue in principle, and generally works as expected, the query planner of PostgreSQL runs into the usual problem with large queries and fails to recognize that semi-joins are possible here. Instead, PostgreSQL chooses join operations, which leads to a blow-up of intermediate results, same as with the original query, and consequently PostgreSQL runs out of time. In fact, the problem that the query planner decides

[2]Note that assuming 20 minutes running time here generally benefits the baseline systems as they produce significantly more timeouts and the actual time to execute those queries is often significantly beyond 20min.

against semi-joins and uses joins instead also appears in the full enumeration case and occasionally also with Spark SQL. As a mitigation, one could adapt the rewriting to perform the semi-joins one after the other in such cases. However, we refrained from doing so since we wanted to provide a portable system that allows us to compare in a uniform way the general feasibility of the structure-guided methods over a wider variety of existing systems. Naturally, deeper integration of structure-guided methods into a DBMS would immediately eliminate such problems.

0MA Aggregation Queries

Method	Timeouts	Max[1]	Mean[2]	Med.[2]	Std.Dev.[2]
DuckDB	58	1169.38	217.9	0.44	447.94
DuckDB+YanRe	**0**	15.57	2.31	1.44	2.38
PostgreSQL	91	1131.08	342.78	2.82	524.16
PostgreSQL+YanRe	**2**	236.75	24.74	5.83	93.73
SparkSQL	91	1082.58	365.76	25.35	518.7
SparkSQL+YanRe	**3**	214.04	41.12	16.14	113.24

Full Enumeration Queries

Method	Timeouts	Max[1]	Mean[2]	Med.[2]	Std.Dev.[2]
DuckDB	69	770.55	252.27	0.67	473.87
DuckDB+YanRe	**29**	801.79	121.39	2.34	335.75
PostgreSQL	97	1107.66	364.32	4.02	533.47
PostgreSQL+YanRe	**70**	786.31	283.2	25.71	470.2
SparkSQL	87	1164.06	358.28	23.91	513.67
SparkSQL+YanRe	**29**	876.74	204.11	59.45	335.47

[1] Excludes timeout values.
[2] Timeout treated as 1200 seconds.

Table 4.1: DuckDB, PostgreSQL, and Spark SQL with or without YanRe for ACQs over the MusicBrainz dataset. All times are reported in seconds.

In Figure 4.2, we provide histograms of how many queries could be executed within certain time brackets, with brackets of $t \leq 1$, $1 < t \leq 10$, $10 < t \leq 100$, $100 < t \leq 1000$ seconds (represented by their upper bounds in the figure). Additionally, we also list the number of queries that timed out (TO).

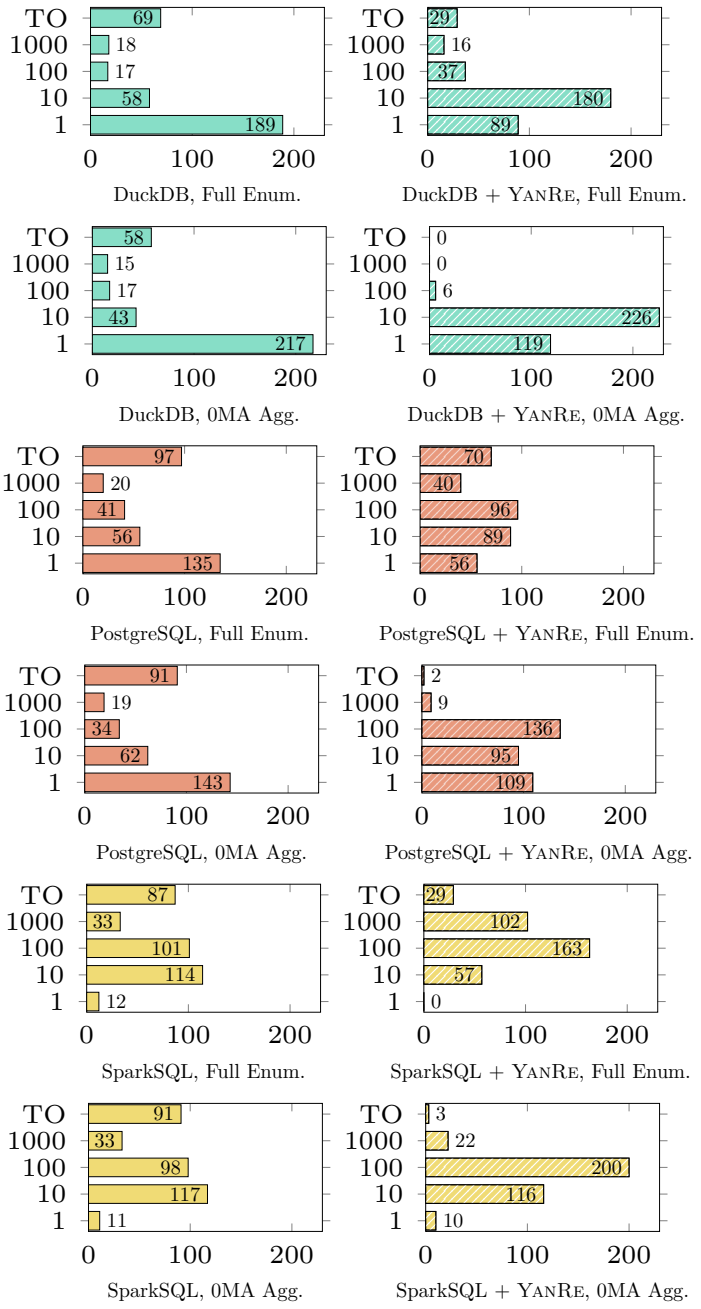


Figure 4.2: Histograms showing how many instances were solved in each time range, with or without YanRe, for the three systems studied.

In order to simplify the presentation, we thus ignore in total 5 runs that took between 1000 and 1200 seconds. The left column of histograms represents the baseline systems. We see a trend of queries being either easy or very difficult for a system, with especially

the large bracket of times between 100 and 1000 seconds being the least common (in particular, for DuckDB and PostgreSQL). The histograms also give a better insight into the improvements achieved through the use of YanRe. With DuckDB, we see that many of the queries causing a timeout with the baseline system can be solved far below the timeout threshold with YanRe, even in the full enumeration case. At the same time, due to the overhead of YanRe, the number of queries that are solved in under a second is significantly lower. For PostgreSQL and Spark SQL we see that the overhead and the aforementioned issues around planners avoiding semi-join operations cause a general trend towards slower evaluation in full enumeration queries, despite significant reduction in timeouts.

Figure 4.3 provides a breakdown of the average time (in seconds) spent in each of the four stages of the YanRe rewriting (excluding timeouts). We have omitted Spark SQL in this figure, since there we have applied there a different approach of executing all stages as one query plan.

The *Setup* phase consists of the creation of various views that represent the initial relations for each node in the join tree. Surprisingly, this takes up a noticeable amount of time in some cases (we later eliminate this overhead through the integration into Systems in Chapter 5). In the case of full enumeration queries, we see for both, DuckDB and PostgreSQL, that YanRe spends the most time in the join phase. It is interesting to note that PostgreSQL also spends a lot of time in the two semi-join phases, whereas for DuckDB, the time spent there is insignificant relative to the Join phase. As we discussed earlier, we have seen cases where the query planner of PostgeSQL eschews the use of semi-joins, which explains parts of this marked difference in the time distribution. Additionally, the handling of internal tables and possible bottlenecks in their creation are another potential factor for this discrepancy. In the 0MA aggregation case, we see that both systems fare very similarly, with DuckDB again requiring more time for the Setup stage. The increase in Setup time over the full enumeration case here is due to the larger number of instances that could be solved without timeout for 0MA queries. Note that in the 0MA case, the "Join" phase consists only of the final aggregation in the root node, which explains the (almost) 0 time consumption.

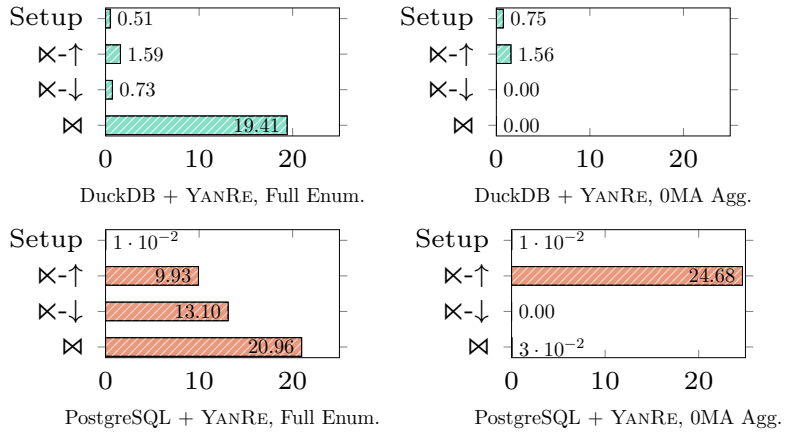


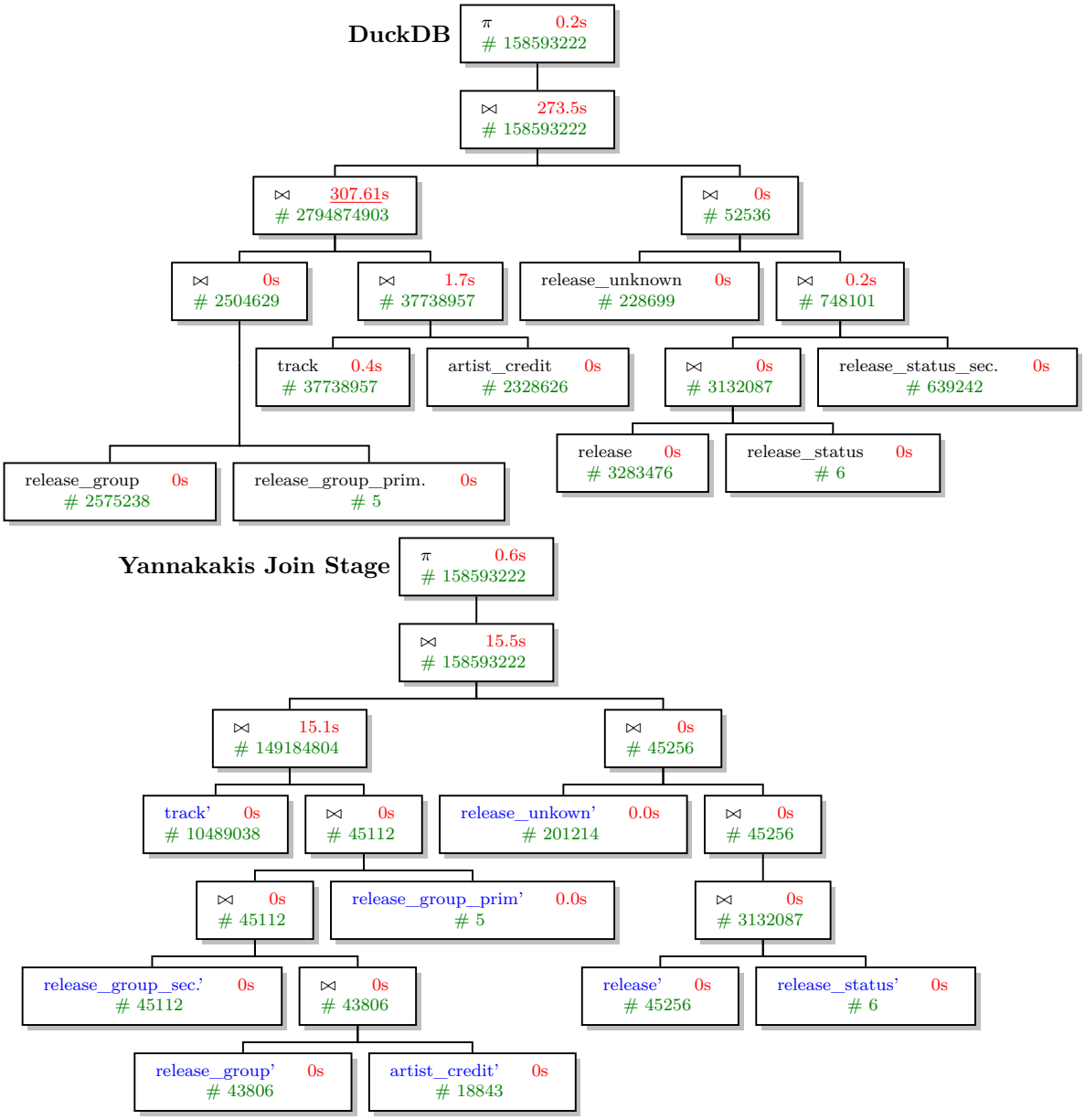
Figure 4.3: A breakdown of the average time in seconds spent by DuckDB and PostgreSQL in each of the four stages of YanRe, corresponding to the 4 phases of Yannakakis' algorithm.

YanRe Planning Time. The time required by YanRe to create the rewriting is negligible even in our unoptimised proof-of-concept implementation. Even for the largest queries (30 relations), the computation of hypergraphs and join trees as well as the subsequent rewriting requires only a few milliseconds. This is magnitudes faster than usual planning times by the host systems for complex queries and we therefore do not provide a more detailed analysis of our planning time here. Detailed records of the times spent by YanRe in the various phases of query execution are available in the aforementioned repository of data and code artifacts.

4.3.3 Deeper Insight into Improvements

We see that structure-guided query evaluation can significantly improve the performance of widely used DBMSs on difficult queries, even if all joins are along foreign key relationships. In this section, we further illustrate the reasons for these improvements in detail.

We consider the evaluation of benchmark query 08ad (for the full enumeration case), which is illustrated in Figure 4.4. On the left-hand side, we show the query plan (projections at leaf nodes are omitted in the figure) as produced by DuckDB on the input query. On the right-hand side, we show the query plan that was produced by DuckDB for the final Join stage query in the YanRe rewriting. That is, all relations at this point have been reduced by the two semi-join passes. To emphasise this, we refer to the reduced version of each relation R as R' in the right tree and mark it in blue. The size of each relation is given in green after a #, and the times in the nodes represent total CPU time (note that this differs strongly from wall clock time due to heavy parallelisation) spent on this operation.



System / Stage	Execution Time
DuckDB	151.9 s
DuckDB + YanRe / Total	42.1 s
DuckDB + YanRe / Setup	0.5 s
DuckDB + YanRe / ⋉-up	1.5 s
DuckDB + YanRe / ⋉-down	0.9 s
DuckDB + YanRe / Join	39.2 s

Figure 4.4: Details of performance difference in query plans of query 08ad.

The query produces a large number of output tuples (≈ 158 million). However, while our rewriting still has to materialise all of these tuples (at significant computational cost), the baseline query plan produces an even larger and more costly intermediate result with ≈ 2.8 billion tuples on the way to the final output. Actually, the huge discrepancy between the original vs. reduced relations is already seen at the leaf nodes of the two query plans: for instance, when we look at the relations artist_credit, release, and release_group, the reduction in size is by a factor of 123, 72, and 58, respectively.

The table at the bottom of the figure provides the wall clock times for evaluation of the baseline using only DuckDB, as well as DuckDB+YanRe. The baseline plan on the left required 151.9 seconds, while our approach took 42.1 seconds to execute. Notably, we see that the significant improvement in the join phase comes at a very cheap cost: the two semi-join phases that allowed us to avoid the blow-up required only a total of 2.4 seconds. Thus, while the query is still solvable in reasonable time in the baseline case, we see that even such cases can be significantly improved by a structure-guided approach.

We note that this query has only 8 relations and the planning phase is therefore still manageable in the baseline case. Specifically, PostgreSQL manages to answer the query in 64 seconds, while only Spark SQL times out. Importantly, even if all joins follow foreign key relationships, there can still be an enormous blow-up of intermediate results if an evaluation strategy based solely on joins (without using semi-joins to remove dangling tuples first) is applied. Advancements in cardinality estimation, which aim at the computation of good join plans, are therefore inherently insufficient on these types of challenging queries.

Indexes. Indexes have traditionally been an important factor in fast join evaluation in DBMSs. However, when the time to evaluate a query is dominated by efforts related to large intermediate results, indexes are of little to no help as they cannot decrease the size of a join. This observation is also confirmed by our experiments with three different DBMSs, which apply significantly different indexing strategies and yet yield comparable experimental results. In PostgreSQL, it is common to maintain a large number of explicitly specified and materialised indexes for all attributes that are deemed important. In our experiments for PostgreSQL we use all indexes that are set in the Musicbrainz dataset, which are effectively on all attributes over which joins are made in our queries. In contrast, Spark SQL supports no indexes at all and DuckDB does not allow persistent indexes (every new session requires a new creation of indexes), but internally maintains ad-hoc index structures for commonly accessed values and attributes. Our experiments therefore run without explicitly declared indexes on both systems[3]. Despite these differences, we see consistent improvements using YanRe over all systems. Furthermore, PostgreSQL performs worst in every measure despite the most elaborate support of indexes among the 3 systems tested here.

[3]Creating all indexes in DuckDB takes over 30 minutes on our test system and it was infeasible to add this overhead to every tested query. Additional experiments showed that explicitly creating the same indexes in DuckDB as in PostgreSQL makes no significant difference to our measured times.

4.3.4 Cyclic Queries

Query	Ordered Eval. Time by GHD (s)							
09ac	10.3	16.5	18.4	t/o	t/o	t/o	t/o	—
11ag	11.2	26.8	t/o	t/o	t/o	t/o	t/o	t/o
11al	6.2	6.3	8.3	258	t/o	t/o	t/o	t/o

Table 4.2: Run times of cyclic queries with different GHDs

To explore how a structure-guided approach to query processing generalises beyond ACQs, we have carried out some preliminary experiments with a few cyclic queries from the benchmark of [110], which we briefly discuss next. In Table 4.2, we show some of these results: we have chosen 3 of the smallest cyclic queries from the benchmark (called 09ac, 11ag, and 11al). As is indicated by their names, these queries involve the join of 9 resp. 11 relations. For each of these queries, we have computed 8 different generalized hypertree decompositions (GHDs) of width 2, which is optimal in these cases. Actually, for 09ac, we were only able to find 7 distinct GHDs. These GHDs were constructed by repeated execution of the decomposition tool BalancedGo [65] with randomised search order. For each of the distinct GHDs computed in this way, we then proceed as in the acyclic case, with the only difference being that the initial relation associated with a tree node u may now either be a base relation or a view obtained by joining the relations of the edge cover labelling the node of the GHD. Turning the GHDs into join trees by carrying out the local joins at each node of the GHD and applying our YanRe system on DuckDB, we obtained the run times (sorted in ascending order) reported in Table 4.2. Without YanRe, the corresponding run times of DuckDB are timeout (query 09ac), 22.22s (query 11ag), and 263.87s (query 11al), respectively. For all queries, we notice a striking discrepancy in execution times of DuckDB + YanRe depending on the chosen GHD: in the best case, DuckDB + YanRe may be way faster than plain DuckDB, in the worst case, DuckDB + YanRe times out.

We see in Table 4.2, that the effort of structure-guided query evaluation via GHDs can vary heavily, depending on the chosen GHD and, in particular, on the joins required to turn the GHD into a join tree. Importantly, even small hypergraphs can have a relatively large number of different GHDs of minimal width. We are therefore confronted with another optimisation problem of finding the GHD with the most efficient reduction to the acyclic case.

We further illustrate this by taking a closer look at one of the cyclic queries thus studied, namely query `09ac`, which we recall in full in Figure 4.5. On the left-hand side of Figure 4.6, we have the hypergraph of this query. For our purposes, only the structure of the hypergraph is relevant and not the precise names of the attributes. For the sake of better readability, we have therefore abbreviated the attribute names to a,b,c,d,e,f. Moreover, attributes irrelevant to the query have been omitted altogether. The correspondence between these abbreviations and the true attribute names is shown

in Table 4.3. In this table, we have omitted the relations which only occur with a single attribute in the query. The correspondence between abbreviation and true name is obvious in these cases: artist_credit.id (abbreviated to a), release_country.release (abbreviated to c), release_group_secondary_type_join.release_group (abbreviated to b), and release_group_prior_type.id (abbreviated to e). Note that we have omitted unary edges (which correspond to relations with a single (relevant) attribute) from the hypergraph since they have no effect on the acyclicity of a query. Of course, in the GHDs, the unary relations have to be reintroduced. However, the join with a unary relation trivially degenerates to a semi-join. Hence, they can never lead to a blow-up of intermediate results.

On the right-hand side of Figure 4.6, we have three of the different GHDs generated for this query in our experiments together with the overall execution time of DuckDB + YanRe to answer the query. For space reasons, the labels of the nodes contain abbreviations of relation names. The correspondence between these abbreviations and the true relation names are shown in Table 4.4. We can observe clear structural differences between the GHDs, with decomposition **Fast** branching only to at most 3 children, while decomposition **Timeout** is flat and very wide. More importantly, the joins needed to turn the GHDs into join trees are markedly different. Decomposition **Timeout** induces the costly cross product between medium and release_group, while decomposition **Fast** avoids such views. The third decomposition **Fast-2** shows a third GHD for which execution is even faster than for **Fast**. Notably, **Fast-2** requires only 2 joins to turn the GHD into a join tree – in contrast to the 5 joins needed in **Fast**. For reference, "plain" DuckDB (i.e., without the rewriting done by YanRe) times out on this query and PostgreSQL solves it in 85 seconds.

```
SELECT   track.recording, track.medium, medium.release,
artist_credit.id, release.release_group,
         release_group.type
FROM     artist_credit, recording, release_group,
release_group_secondary_type_join,
release_group_primary_type, track, release,
medium, release_country
WHERE    artist_credit.id = recording.artist_credit
        AND release.id = medium.release
         AND artist_credit.id = release_group.artist_credit
         AND track.medium = medium.id
         AND release_group.id = release_group_secondary_type_join.release_group
         AND release.id = release_country.release;
         AND release_group.type = release_group_primary_type.id
         AND artist_credit.id = track.artist_credit
         AND recording.id = track.recording
         AND artist_credit.id = release.artist_credit
         AND release_group.id = release.release_group
```

Figure 4.5: Query 09ac (full enumeration)

To summarise, our preliminary experiments with cyclic CQs show that there is clear potential for structure-guided query answering beyond acyclic queries. But they also show that this requires new methods for finding the "right" decompositions. Indeed, the key observation is that a good choice of decomposition is absolutely crucial for the performance of query evaluation. We will develop solutions for finding such decompositions in Chapter 6.

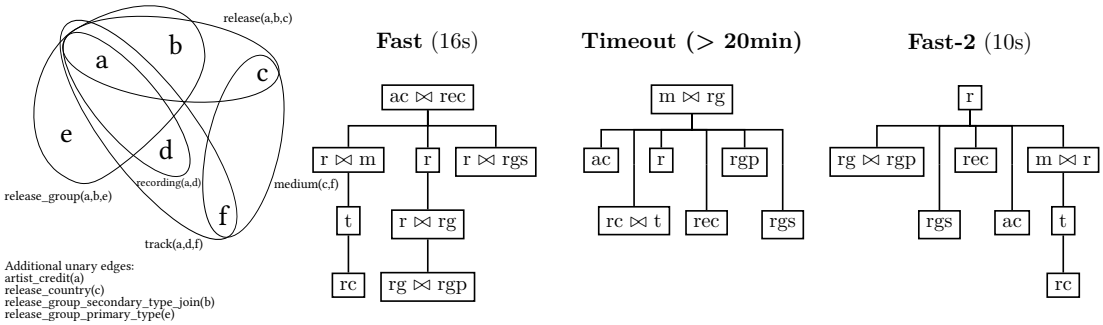


Figure 4.6: Hypergraph and different GHDs of the cyclic query 09ac

relation plus schema	true attribute names
medium(c,f)	release,id
recording(a,d)	artist_credit,id
release(a,b,c)	artist_credit,release_group,id
release_group (a,b,e)	artist_credit,id,type
track(a,d,f)	artist_credit,recording,medium

Table 4.3: Abbreviations of attribute names in query q09ac

abbreviation	true relation name
ac	artist_credit
m	medium
r	release
rc	release_country
rec	recording
rg	release_group
rgp	release_group_primary_type
rgs	release_group_secondary_type_join
t	track

Table 4.4: Abbreviations of relation names in query q09ac

4.4 Query Rewriting as an Algorithm Selection Problem

We have, up to this point, introduced and evaluated the rewriting-based approach, showing its potential for improving query execution on hard instances, but also identifying weaknesses on simpler instances. To make this approach useful in practice, we urgently need to find a way to apply it only whenever it improves performance. To address this problem, we develop a decision procedure that decides when the new method should be applied. We are thus faced with an algorithm selection problem, where we have to decide, for every database instance and query, which query evaluation method should be applied. In this section, we describe the steps needed to formulate the precise algorithm selection problem. An overview of this workflow is given in Figure 4.7.

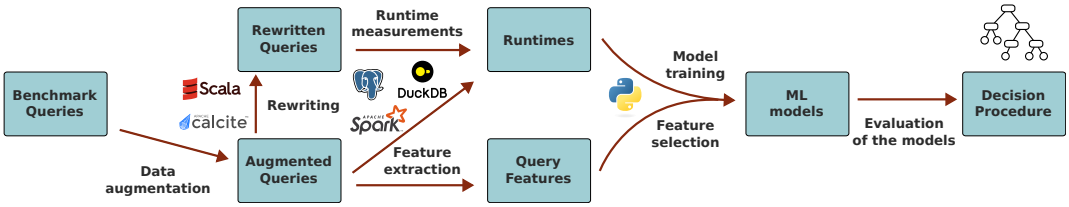


Figure 4.7: Methodology workflow.

It consists of the following steps: We first have to create a suitable dataset that features the required variety and size for the ML task at hand. On one hand, we thus have to (1) *select and adapt common benchmarks* and select those queries to which the optimisation technique is applicable. On the other hand, we (2) *apply data augmentation* to ensure a suitable size of the dataset. We then need to (3) *select DBMSs* on which we want to test the effectiveness of the new optimisation techniques. Next, we (4) *rewrite the given SQL queries* into sequences of SQL commands that "force" the selected DBMSs into a particular evaluation strategy. To prepare for the ML task, we then have to (5) *do a feature selection*. That is, we want to characterise every query in terms of a feature vector. We are then ready to (6) *run the experiments*, i.e., we execute all queries with and without the optimisation and measure the times of each run. The result will be runtimes for each query (characterised by a particular feature vector) both, when the optimisation is applied and when it is not applied. This is then the input to the *model training step* and, ultimately, to the development of a decision procedure, which will be described in Section 4.5.

4.4.1 Benchmark Data

To construct our new dataset *MEAMBench* (**M**aterialisation **E**xplosion **A**ugmented **M**eta **B**enchmark), we pursue two major goals to make the dataset suitable for model training and testing: it should be sufficiently big and diverse. To address the diversity aspect, we collect datasets and queries from different domains, designed for different purposes, and representing challenging cases of an explosion of intermediate results and materialisation. Thus, as a basis, we have chosen several widely used benchmarks, which contain join queries over several relations: (1) The *JOB (Join Order Benchmark)* [101], which was introduced to study the join ordering problem, is based on the real-world IMDB dataset and contains realistic join-aggregation queries with many joins and various filter conditions, (2) *STATS/STATS-CEB* [76] is based on the Stackexchange-dataset, and contains many join queries not following FK/PK-relationships, (3) Four different datasets (namely cit-Patents, wiki-topcats, web-Google and com-DBLP) from *SNAP (Stanford Network Analysis Project)* [103], a collection of graphs, which we combine with synthetic queries introduced in [99], (4) *LSQB (Large-Scale Subgraph Query Benchmark)* [116], which was designed to test graph databases as well as relational databases, consists of synthetic data and hard queries based on a social network scenario, and (5) *HETIONET* [80]. The latter is less known in the database world. It contains real-world queries on real-world

data from a heterogeneous information network of biochemical data, and is part of the new CE benchmark[40], which has, for instance, been recently used in [23] and [20].

Again, we focus on ACQs, which make up the vast majority of the queries in the base benchmarks. Most of the queries in the chosen benchmarks are CQs with additional filter conditions applied to single tables. These filter conditions can be taken care of by the preparatory step; so they pose no problem. However, not all the CQs are acyclic; so we have to eliminate the cyclic ones from further consideration. The number of (acyclic) CQs of each dataset is given in Table 4.5.

Note that some of the queries in the benchmarks are enumeration queries and already contain some aggregate (in particular, `MIN`) and satisfy the 0MA conditions. Of course, also from the enumeration queries, we can derive 0MA queries by putting an appropriate aggregate expression (again, in particular, with the `MIN` aggregate function) into the SELECT clause of the query. We do this by randomly choosing a table occurring in the query and one column of this table. We will see in Section 4.4.2, that it makes no significant difference which table and attribute we choose for turning a query into 0MA form, as we will vary the table and attribute anyway.

4.4.2 Data Augmentation

Our collection of data and queries from different benchmarks results in 219 acyclic queries, as can be seen in Table 4.5. Since our goal is to use our new dataset MEAMBench for training and testing ML models, this is clearly not a sufficient amount. Therefore, we perform data augmentation, as will be detailed next.

Dataset	**# ACQs**	**+filter**	**+filter&agg**	**+filter&enum**
STATS	146	432	1876	1264
SNAP	40	40	244	120
JOB	15	45	264	135
LSQB	2	2	14	6
HETIONET	26	72	538	216
Total	219	591	2936	1741

Table 4.5: Overview of the 0MA and enumeration queries after augmentation. In total, we get 4677 queries, consisting of 2936 0MA queries and 1741 enum queries.

For our dataset, we decided to use the following two steps for data augmentation: "filter augmentation" (for all queries) followed by "aggregate-attribute augmentation" (for 0MA-queries) and “enumeration augmentation” (for enumeration queries), respectively.

With the filter augmentation, we want to get duplicates of all queries having filters (i.e., selection conditions on a single table) and then change some filters in a way that the sizes of the resulting relations vary between these queries. If the query had only one filter,

we change the specific value it is equal to, greater or smaller of the filter condition. For these cases, we get twice as many queries as before. For all queries having two or more filters we choose two filters, which we change, each at a time. Here we try to replace the filters in a way that once the number of answer tuples gets bigger and once smaller. This gives us triples for each of these queries.

Example 4.4.1. *Consider the STATS query '005-024', named q here. To illustrate the filter augmentation, we present two possible augmentations on q. One option to augment q is to swap the filter condition,* `v.BountyAmount>=0` *, with a transformed one, such as* `v.BountyAmount>=40`*, producing the query* q_{aug1}*. Another option is to swap* `u.DownVotes=0` *with* `u.DownVotes=10`*, producing* q_{aug2}*.*

```
q:    SELECT MIN(u.Id)
      FROM   votes AS v, badges AS b, users AS u
      WHERE  u.Id = v.UserId AND v.UserId = b.UserId
           AND v.BountyAmount>=0 AND v.BountyAmount<=50
           AND u.DownVotes=0
```

```
q_aug1: SELECT MIN(u.Id)
      FROM   votes AS v, badges AS b, users AS u
      WHERE  u.Id = v.UserId AND v.UserId = b.UserId
           AND v.BountyAmount>=40 AND v.BountyAmount<=50
           AND u.DownVotes=0
```

```
q_aug2: SELECT MIN(u.Id)
      FROM   votes AS v, badges AS b, users AS u
      WHERE  u.Id = v.UserId AND v.UserId = b.UserId
           AND v.BountyAmount>=0 AND v.BountyAmount<=50
           AND u.DownVotes=10
```

For 0MA queries, we next apply the "aggregate-attribute augmentation" to vary the table from which we take the MIN-attribute. This is done in a way that every table occurring in the query appears once in the MIN-expression. The column of the chosen table does not really matter, which means we just take the first column of the table. Depending on the number of tables involved in the query, this leads to a different number of new queries per query.

Example 4.4.2. *We give an example for the aggregate-attribute augmentation on 0MA queries. As in Example 4.4.1, we again focus on the STATS query 005-024. For this query, we thus create 3 versions by taking either* `MIN(u.Id)`*,* `MIN(v.Id)`*, or* `MIN(b.Id)` *in the SELECT clause. This aggregate-attribute augmenation is applied to the original queries and to the filter augmented ones alike. Hence, the original STATS query 005-024 gives rise to 9 distinct queries after the whole augmentation process.*

```
005-024: SELECT MIN(v.Id)
         FROM votes as v, badges as b, users as u
```

```
        WHERE u.Id = v.UserId AND v.UserId = b.UserId
            AND v.BountyAmount>=0 AND v.BountyAmount<=50
            AND u.DownVotes=0
005-024-augA1: SELECT MIN(b.Id)
        FROM votes as v, badges as b, users as u
        WHERE u.Id = v.UserId AND v.UserId = b.UserId
            AND v.BountyAmount>=0 AND v.BountyAmount<=50
            AND u.DownVotes=0
005-024-augA2: SELECT MIN(u.Id)
        FROM votes as v, badges as b, users as u
        WHERE u.Id = v.UserId AND v.UserId = b.UserId
            AND v.BountyAmount>=0 AND v.BountyAmount<=50
            AND u.DownVotes=0
```

In Table 4.5, we summarize the numbers of 0MA queries that we get after each step of the augmentation. The SNAP and LSQB queries do not have filter conditions, which means there is no filter augmentation for them.

We also take the enumeration queries, for which filter augmentation has already been done, and apply an enumeration augmentation step. To this end, we randomly choose two of the attributes used in join conditions and write them into the SELECT clause of the query. This is done three times for each filter augmented query if at least three different join attributes exist in the query. On the other hand, a query with only one join gives rise to only a single enumeration query (with the join attributes in the SELECT clause) in our dataset.

In summary, after applying the data augmentation step to the 0MA and enumeration queries, we have 4677 queries in total.

4.4.3 Selection of DBMSs

We aim to evaluate the effectivity of the optimisation via Yannakakis-style query evaluation on a wide range of database technologies. Therefore, we have chosen three significantly different DBMSs, namely (1) PostgreSQL 13.4 [146] as a "classical" row-oriented relational DBMS, (2) DuckDB 0.4 [134] as a column-oriented, embedded database, and (3) Spark SQL 3.3 [164] as a database engine specifically designed for distributed data processing in a cluster. These DMBSs represent a broad spectrum of architectures and characteristics and they, therefore, give a good overview of the range of existing DBMSs.

4.4.4 Query Rewriting

For query rewriting, we again make use of the techniques described at the beginning of the chapter, where a single SQL query is rewritten into an equivalent series of queries, to guide DBMSs to utilise a given optimisation method. The implementation is based on work in [24], and Scala code is partially derived from the integration into Spark SQL described in Chapter 5.

4.4.5 Feature Selection

We choose different kinds of features that we derive from the structure of the query itself, from the join tree constructed in the process of rewriting the query, and from statistics determined by PostgreSQL or DuckDB over the database. The latter kind of features is extracted from the query optimiser's estimates, and obtained via the EXPLAIN command. Note that Spark SQL does not provide an EXPLAIN command. However, we will explain below how to circumvent this shortcoming. Another challenge are features that are based on a set, of variable length, containing numeric values. In order to reduce such a set of values into a fixed-length list of values, we calculate, for each set, several statistics: min, the 0.25-quantile (referred to as q25), median, the 0.75-quantile (referred to as q75), max, and mean. In the list of features below, we use * (e.g. **B7***) to mark which features consist of variable-length sets, and hence will get reduced to the mentioned collection of 6 values.

Features derived from the query. The following features are easily obtained by inspecting the query itself:
Feature B1: *is 0MA?* indicates (1 or 0) if the query is 0MA,
Feature B2: *number of relations*,
Feature B3: *number of conditions*, which refers to the number of (in)equality conditions in the WHERE clause of the query,
Feature B4: *number of filters*, which more specifically only counts the (in)equality conditions occurring in the query, and
Feature B5: *number of joins*.

Features based on the join tree. The following features are inspired by the work in [4] on tree decompositions:
Feature B6: *depth*, which is the maximal distance between the root of the used join tree and a leaf node,
Feature B7*: *container counts*, which is a set of numbers, indicating for each variable in the query the number of nodes in the join tree it occurs in. This measure indicates how many relations are joined on the same variable, and
Feature B8*: *branching degrees*, which is a set of numbers, indicating for each node the number of children it has.

These eight features (B1)-(B8*) are the shared *"basic features"*. In addition, we can use statistical information from the database and the estimates for the query evaluation, though the exact features that are exposed differs between DBMSs. In case of PostgreSQL and DuckDB, we have the EXPLAIN command at our disposal to obtain relevant further information. For PostgreSQL, we thus select the following additional features, which we refer to as *"PSQL features"*:
Feature P1: *estimated total cost* (of the query),
Feature P2*: *estimated single table rows*, which stands for the estimated number of rows for each table involved in the query *after* the application of the filter conditions, and

Feature P3*: *estimated join rows*, the estimated number of rows of each join *before* the application of the filter conditions.

The EXPLAIN command for DuckDB behaves differently. It allows us to derive a single *"DDB feature"*:
Feature D1*: *estimated cardinalities*, the estimated number of rows after each node in the logical plan, such as filters and joins.

As SparkSQL does not perform cost-based optimisation, it also does not have any statistical information of the data, and cannot estimate cardinalities or costs of a plan. However, since SparkSQL also does not provide a persistent storage layer, tables are commonly imported from another database via JDBC. This implies that, in practice, the statistical features can easily be extracted from this database and used for the decision whether to rewrite the query in SparkSQL. For the experiments presented in Section 4.6, we extracted these features from PostgreSQL, hence SparkSQL will have the same feature set as PostgreSQL.

4.4.6 Running the Queries

The whole evaluation is performed on a server with 128GB RAM, a 16-core AMD EPYC-Milan CPU, and a 100GB SSD disk, running Ubuntu 22.04.2 LTS. After a warm-up run, the original query, as well as the rewritten version, is evaluated five times, and then we take the mean of those five runtimes. In total, we get 6 data points for each of the 4677 queries: each query is run against 3 DBMSs, where the query evaluation happens once with and once without optimisation. Aggregated information on these runtimes is provided in Section 4.6.

In addition to the six runtimes, we also add a "feature vector", consisting of the features described in Section 4.4.5. These provide the input to the training of ML models to be described next.

4.5 Solving the Algorithm Selection Problem

Several decisions have to be made to solve the algorithm selection problem that results from the steps described in the previous section. In particular, we have to (1) *formulate the concrete ML problem* and then (2) *select ML model types* together with hyperparameters appropriate to our context. Before we can start training and testing the models, we have to (3) *split the data* (in our case the SQL queries) into training/validation/test data. Finally, we have to (4) *define selection criteria* for determining the "best" model, which will then be used as basis of our decision procedure between the original query evaluation method of each system and a Yannakakis-style evaluation.

4.5.1 Formulating the ML Problem

Our ultimate goal is the development of a decision procedure between two evaluation methods for acyclic queries. Hence, we are clearly dealing with a *classification problem* with *2 possible outcomes*. In the sequel, we will refer to these two possible outcomes as 0 vs. 1 to denote the original evaluation method of the DBMS vs. a Yannakakis-style evaluation (enforced by our query rewriting).

On the other hand, it also makes sense to consider a *regression problem* first and, depending on the predicted value, classify a query as 0 (if the predicted value suggests faster evaluation by the original method of the DBMS within a certain threshold) or 1 (otherwise). As the target of the regression problem, we would like to choose the difference $t_{\text{rewritten}} - t_{\text{original}}$, where we write $t_{\text{rewritten}}$ and t_{original} to denote the time needed by Yannakakis-style evaluation and by the original evaluation method of the DBMS, respectively. However, as will become clear in our presentation of the experimental results in Section 4.6, the actual runtime values are very skewed, in the sense that their distribution shows high variance. Hence, the difference we focus on is also highly skewed. To get more reliable results, we therefore perform a (variant of) log-transformation as described next: Since we may have negative values, we cannot apply the logarithm directly. Instead, we multiply the log of the absolute values with the sign they had before. Additionally, since we have a lot of values close to zero (which leads to very small log values) we add 1 to the absolute values before applying the log, which is a common method in such situations. The transformation therefore results in the following formula: $x_{new} = sgn(x) * log(|x| + 1)$. In Figure 4.8, we can see this difference function under log transformation over the data from our experiments shown in Section 4.6.

4.5.2 Selecting Model Types

We have chosen 7 Machine Learning model types for our algorithm selection problem, namely k-nearest neighbours (k-NN), decision tree, random forest, support vector machine (SVM), and 3 forms of neural networks (NNs): multi-layer perceptron (MLP), hypergraph neural network (HGNN) and a combination of the two. MLP is the "classical" deep neural network type. Hypergraph neural networks, introduced in [53], are less known. With their idea of representing the hypergraph structure in a vector space, HGNNs seem well suited to capture structural aspects of conjunctive queries. Just like MLPs, also the HGNNs produce an output vector. In our combination of the two model types, we provide yet another neural network, which takes as input the two output vectors produced by the MLP and the HGNN and combines them to a joint result using additional layers.

A major task after choosing these ML model types is to fix the hyperparameters. An overview of some basic hyperparameters is shown in Table 4.6. Of course, in particular for the 3 types of neural networks, many more hyperparameters have to be fixed.

Model	Hyperparameters
Random Forest	#estimators = 100
kNN	k = 5
SVM	kernel = linear
MLP	layers = 30-60-40-2
HGNN	layers = 1-16-32-2

Table 4.6: Chosen hyperparameters.

4.5.3 Labelling and Splitting the Data

After running the 4677 queries mentioned in Section 4.4.2 on the 3 selected DBMSs according to Section 4.4.3, we have to prepare the input data for training the ML models of the 7 types mentioned in Section 4.5.2. Recall from Section 4.4.5 that each query is characterised by a feature vector specific to each of the 3 DBMSs. For our supervised learning tasks (classification and regression), we have to label each feature vector for each of the 3 DBMSs. As explained in Section 4.5.2, we want to train our models both, for classification and for regression. Hence, on the one hand, each feature vector gets labelled 0 or 1 (meaning that the original evaluation of the DBMS or the Yannakakis-style evaluation is faster; in case of a tie, we assign the label 0) for the classification task. On the other hand, each feature vector is labelled with the difference of the runtime of the original evaluation minus the runtime of the Yannakakis-style evaluation for the regression task.

The labelled data can then be split into training data, validation data, and test data. In principle, we choose a quite common ratio between these three sets by letting the training set contain 80% of the data and the other two contain 10% each. However, to get more accurate results, we have decided to do 10-fold cross validation. That is, we split the 90% of the data that were chosen for training and validation in 10 different ways in a ration 80:10 into training:validation data and, thus, repeat the training-validation step 10 times.

4.5.4 Model Selection Criteria

In order to ultimately choose the "best" model for our decision procedure between the original evaluation method of each DBMS and the Yannakakis-style evaluation, we compare, for every feature vector, the predicted classification with the actual labelling. We refer to classification 1 as "positive" and classification 0 as "negative". This leads to 4 possible outcomes of the comparison between predicted and actual value, namely TP (true positive) and TN (true negative) for correct classification and FP (false positive) and FN (false negative) for misclassification. They give rise to the 3 most common metrics: accuracy (shortened to "Acc"), which is the proportion of correct classifications, precision (shortened to "Prec"), which is defined by TP / (TP + FP), and recall (shortened to "Rec"), defined as TP / (TP + FN).

Of course, the natural goal when selecting a particular model is to maximize the accuracy. However, in our context, we consider the precision equally important. That is, we find it particularly important to minimise false positives, i.e., in case of doubt, it is better to stick to the original evaluation method of the DBMS rather than wrongly choosing an alternative method.

For regression, we aim at minimising the mean squared error (MSE). But ultimately, we also map the (predicted and actual) difference between the runtime of the original minus Yannakakis-style evaluation to a 0 or 1 classification. Hence, we can again measure the quality of a model in terms of accuracy, precision, and recall.

Apart from the purely *quantitative* assessment of a model in terms of accuracy, precision, and recall, we also carry out a *qualitative* analysis. That is, for each of the misclassified cases, we want to investigate by how much the chosen evaluation method is slower than the optimal method. And here, we are again particularly interested in the false positive cases. Apart from aiming at high accuracy and precision, we also want to make sure for the false positive classifications, that the difference in the runtimes between the two evaluation methods is rather small.

4.6 Experimental Results

In this section, we present experimental results obtained by putting the algorithm selection method described in Sections 4.4 and 4.5 to work. We thus first evaluate in Section 4.6.1 the performance of various machine learning models on the raw dataset of query runtimes obtained for the selected and augmented benchmarks on the chosen DBMSs. Afterwards, in Section 4.6.4, we evaluate the performance gains by combining the best algorithm selection model with our rewriting method for evaluating acyclic queries. In particular, we perform experiments to answer the following key questions:

Q1 How well can machine learning methods predict whether Yannakakis-style query evaluation is preferable over standard query execution?

Q2 Can we use these machine learning models to gain insights about the circumstances in which Yannakakis-style query evaluation is preferable over standard query execution?

Q3 How well does good algorithm selection performance translate to query evaluation times on different DBMSs?

Q4 To what extent can we optimise for precision while maintaining end-to-end runtime and accuracy?

The source code of the implementation, information on running the benchmarks and model training, as well as the data presented here can be found in the following repository:

https://github.com/dbai-tuw/yannakakis-rewriting. The MEAMBench dataset can be found in the following repository: https://github.com/dbai-tuw/MEAMBench

We also note that the repositories includes various tools, such as Jupyter Notebooks, making it easily possible to reproduce the experiments, including the training of all tested machine learning models.

In the remainder of this section, we present our experimental results and a discussion as to how they answer our key questions.

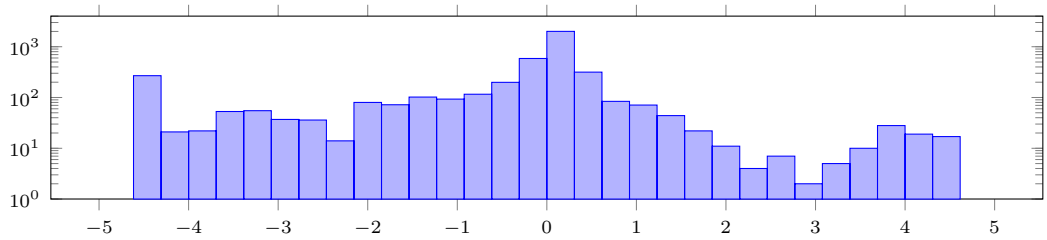
Algorithm	**Acyc. Queries**				
	MSE	MAE	Acc.	Prec.	Rec.
Decision Tree	**0.02**	**0.06**	**0.96**	**0.96**	**0.94**
Random forest	0.03	0.08	0.95	0.94	0.93
k-NN	0.17	0.18	0.91	0.88	0.91
SVM	1.02	0.61	0.79	0.76	0.72
MLP	0.39	0.32	0.81	0.72	0.77
HGNN	0.74	0.48	0.71	0.57	0.85

Table 4.7: Performance of Regression Models. We show MSE and MAE for regression models predicting the difference between the runtime of the original and the rewritten query. Additionally, we present the accuracy and precision after converting the regression model to a classification model by setting a threshold at a predicted time difference of 0 seconds.

4.6.1 Model Training

In a first step, we will compare the performance of various learned models in terms of accuracy, precision, and recall. Table 4.8 compares the performance of the best classification models (with the hyperparameters given in Table 4.6 on the 0MA queries only, as well as on all queries (i.e., 0MA and enumeration), on the runtime data from PostgreSQL. Decision trees and random forests, with roughly the same performance, achieve the best accuracy, precision and recall out of all classifiers. kNN achieves the next best performance, although significantly less at 91% accuracy compared to the 95% accuracy of the decision tree-based models. The random forest classifier has a similar performance to the simple decision trees. Hence, its disadvantage of additional complexity is not outweighed by a significant performance improvement. Therefore, the simple decision tree classifier is the clear choice out of all compared.

Next, we compare the performance of the regression variants of these models, presented in Table 4.7 and (applying the transformation described in Section 4.5.1) in Figure 4.8 on all queries, again on the PostgreSQL runtimes. We can see that the regression performance in terms of MSE and MAE corresponds closely to the classification performance of the classification models, with the decision tree and random forest models again at the



$$diff := rewritten - original\_runtime$$
$$sgn(diff)\log(|diff| + 1)$$

Figure 4.8: Distribution of the regression response, understood as the time difference between rewritten runtime and original runtime for PostgreSQL, under the given log transformation. Above we show a histogram of this distribution, using log scaling to allow for more visual clarity.

top. However, the gap to the next best model is even larger, corresponding to a 5-8x increase in MSE and a 2-3x increase in MAE, making the decision tree-based models again preferable in this situation. From these results, it can be observed that we are able to predict the runtime difference (i.e., runtime of the rewritten query minus runtime of the original query) quite accurately. This predicted runtime difference naturally lends itself to classification by choosing the query rewriting if and only if this difference is below 0. This classification derived from the decision tree regression model leads to a similar (actually slightly better by 0.5%) performance than the decision tree classification. Moreover, as will be discussed next, classification based on the predicted runtime difference provides additional flexibility for fine-tuning the trade-off between precision and recall. Hence, we have chosen classification based on the decision tree regression model as basis for our decision procedure.

Algorithm	**0MA Queries**			**Acyc. Queries**		
	Acc. ↑	Prec. ↑	Rec. ↑	Acc. ↑	Prec. ↑	Rec. ↑
Decision Tree	**0.94**	**0.92**	**0.97**	**0.95**	**0.95**	0.92
Random forest	**0.94**	**0.92**	**0.97**	**0.95**	0.94	**0.93**
k-NN	0.91	0.91	0.90	0.91	0.88	0.91
SVM	0.85	0.85	0.84	0.84	0.82	0.77
MLP	0.87	0.89	0.86	0.85	0.84	0.77
HGNN	0.83	0.84	0.85	0.79	0.70	0.75
HGNN+MLP	0.82	0.78	0.93	0.81	0.77	0.72

Table 4.8: Performance of Machine Learning Classifiers on the PostgreSQL runtimes. We show accuracy, precision and recall for binary classifiers that predict whether rewriting to Yannakakis style evaluation leads to performance gain.

Algorithm	**0MA Queries**		**Acyc. Queries**	
	Acc. ↑	Prec. ↑	Acc. ↑	Prec. ↑
Decision Tree	**0.94**	**0.92**	**0.95**	**0.95**
Random forest	0.94	**0.92**	**0.95**	0.94
k-NN	0.91	0.91	0.91	0.88
SVM	0.85	0.85	0.84	0.82
MLP	0.87	0.89	0.85	0.84
HGNN	0.83	0.84	0.79	0.70
HGNN+MLP	0.82	0.78	0.81	0.77

Table 4.9: Performance of Classification Models. We show both accuracy and precision for binary classifiers that predict whether rewriting to Yannakakis style evaluation leads to performance gain.

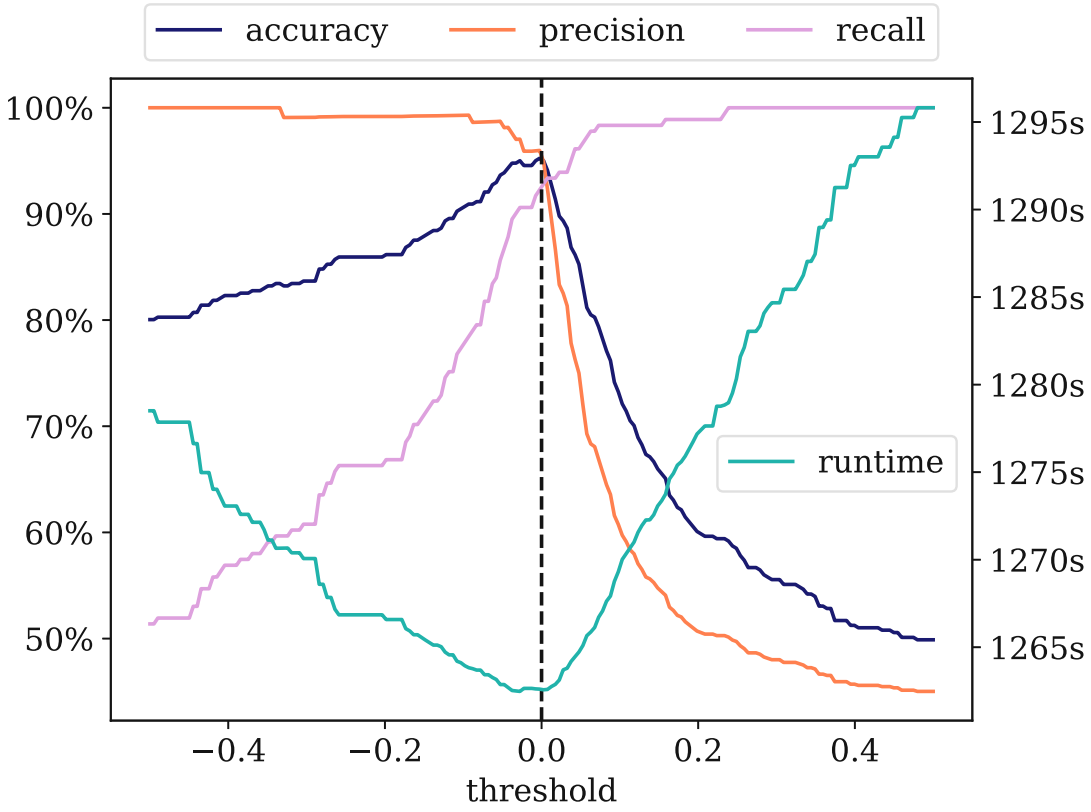


Figure 4.9: Accuracy, precision, recall, as well as the e2e runtime over the test set, of the regression model converted to a classifier, depending on the threshold set (decision tree regression, PostgreSQL, all queries).

4.6.2 Fine-tuning Precision and Recall

So far we have only considered choosing a *threshold* of the predicted runtime difference of the regression model at 0 (i.e., if the predicted runtime difference is below this threshold, we choose Yannakakis-style query evaluation, and, otherwise, we opt for the original query execution). However, we can make use of the regression model as a useful tool to configure the trade-off between precision and recall, depending on the requirements of the application. To simplify the presentation, we focus on one DBMS and on one model, namely PostgreSQL and decision trees. The extension to other models and DBMSs is straightforward and yields similar results. In Figure 4.9, we show how changing the threshold affects the accuracy, precision and recall of the resulting classification model. Clearly, the maximum accuracy is at the 0-threshold. The accuracy, however, continues to be high in the direction of negative thresholds, while falling off quickly in the direction of a positive threshold. In particular, this shows that an optimisation of precision can be performed without sacrificing much recall.

PostgreSQL		DuckDB	
Feature	**Gini**	**Feature**	**Gini**
P3* max(est. join rows)	0.369	**B1** is 0MA?	0.280
B1 is 0MA?	0.157	**B7*** mean(cont. c.)	0.206
P2* q75(est. sing. table rows)	0.069	**D1*** max(est. card.)	0.147
P1 est. total cost	0.053	**D1*** q25(est. card.)	0.061
P3* min(est. join rows)	0.051	**D1*** q75(est. card.)	0.057

Table 4.10: Most important features according to Gini coefficient of the Decision Tree models. The features are described in Section 4.4.5

To summarize our findings on the quality of predicting when Yannakakis-style query evaluation is better: the results from Table 4.8 and Table 4.7 that show very high levels of accuracy, precision and recall for our chosen model of decision trees, as well as the ability to further optimise for precision as seen in Figure 4.9 allow us to positively answer the key question **Q1**. In Figure 4.9 we also get an answer for our key question **Q4**: by choosing the right threshold one can achieve almost perfect precision, with only modest reductions in accuracy and e2e runtime (which we discuss in detail in Section 4.6.4).

4.6.3 Insights from Decision Trees

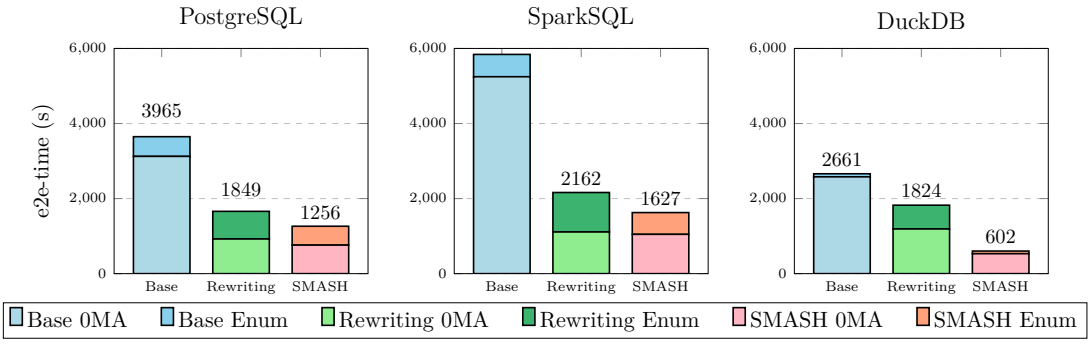


Figure 4.10: Comparison of e2e performance over the test set queries for three database systems: PostgreSQL, Spark (via SparkSQL), and DuckDB. The full bar indicates the runtime over all test set queries, the lower mark indicates the time for 0MA queries.

It is of particular interest that decision trees are among the top-performing models as they are highly interpretable and can provide us with deeper insights into the features that strongly affect the prediction. In Table 4.10, we present the Gini coefficients of the top 5 most relevant attributes, for the PostgreSQL and the DuckDB decision tree classifiers. Note that *the feature set between the two systems is different*, as explained in Section 4.4.5, so we cannot compare the Gini coefficients of all the same features. The Gini coefficient [32] measures the contribution of the feature to the outputs of the model. Looking at the Gini coefficients, we see that, although the performance of the models

trained on the PostgreSQL and DuckDB features and runtimes are very similar, the decision trees rely on different features. In the case of DuckDB, the feature indicating whether the query is a 0MA query has the highest importance. While the PostgreSQL model strongly relies on the maximum join rows in the plan, DuckDB's second-most-important feature is the mean number of container counts, a feature extracted from the join tree. The model for PostgreSQL, on the other hand, only use a single basic feature, namely **B1** (is 0MA?), among the most important features. This ability of decision trees to highlight which specific features affect the prediction the most, helps us answer the key question **Q2** positively. We hope our full experimental artifacts, which include the full decision trees, will help foster further research into using these features to improve query engine optimisation in general.

4.6.4 Effects on Database Performance

So far, we have evaluated purely the performance of the machine learning models on the dataset that we created for our algorithm selection problem. However, our initial hypothesis was that machine learning based algorithm selection can solve the complex challenge of deciding when evaluation in the style of Yannakakis' algorithm is preferable. We therefore now move on to evaluating the performance of the resulting full system that uses our trained algorithm selection models to decide when to rewrite, and thus execute queries using the predicted best query execution method.

We first need to decide which of the two models, the decision tree model or the regression model, we pick for the experiments, since we aim to identify the best algorithm selection mechanism. However, looking at Table 4.7, which shows the performance of the regression model, and Table 4.8, which shows the performance of the classification model, we see that the two are fairly comparable, with a slight advantage for the regression model. As we also saw, the regression-based approach is especially promising for real-world applications as it allows us to fine-tune the decision threshold. We thus choose the regression approach with a threshold at 0 for the analysis presented in this section.

As mentioned in the introduction, we will refer to this integrated method, that decides whether to rewrite to Yannakakis-style evaluation based on the prediction of the decision tree model as SMASH, short for ***S**upervised **M**achine-learning for **A**lgorithm **S**election **H**euristics*. In this section, we will perform all experiments only on the test set queries. *No queries from these experiments were seen by the model at training time, inluding to select the best model.*

At the most fundamental level, we are interested in improving overall query answering performance. We investigate this by analysing the end-to-end (e2e) time necessary to answer all queries in the test set, where "end-to-end" refers to taking the time of running all the benchmark queries and looking at it cumulatively. We note that this also includes the time for the algorithm selection included in SMASH, which was around 2 milliseconds end-to-end.

We summarize our analysis in Figure 4.10, where *Base* refers to the baseline of executing the queries directly in the DBMS, *Rewriting* refers to the time always using the rewriting to Yannakakis-style evaluation, and SMASH refers to the use of our algorithm selection model as described above. To study the robustness of our approach we perform these experiments on three different DBMSs: PostgreSQL, Spark (via SparkSQL), and DuckDB. The significant technical differences between the three systems provide us with a way to study the performance of our method independently of specific DBMS technologies. We report timeouts as follows: if only one of the evaluation methods (rewriting or base case) times out, then we report in Figure 4.10 for such queries as runtime the value of the timeout (= 100s). On the other hand, if both evaluation methods time out, we exclude them from the comparison, since algorithm selection cannot affect anything in such a case. Out of the 441 queries involved in the test set, there were 27 queries that timed out for both evaluation methods on PostgreSQL, 13 queries that timed out for both evaluation methods on SparkSQL, and 30 queries that timed out for both evaluation methods on DuckDB.

Consistently over all systems, we can observe a large improvement over both alternatives by using algorithm selection. Furthermore, we see that even always rewriting overall causes significant improvements over baseline execution of all three systems tested. However, this improvement comes from speedups specifically on queries that are hard for traditional RDBMS execution. These large improvements offset more common minor slowdowns using the *Rewriting* approach. Using SMASH we are able to get the best of both worlds, the major speedups without the minor slower cases.

We conclude that, for our key question **Q3** regarding the effect of the quality of algorithm selection on query evaluation times, we can give a positive answer. Indeed, our algorithm selection clearly improves the e2e query evaluation times across a number of different queries and three different DBMSs.

To provide further insight into how e2e query evaluation performance is affected, Figure 4.11 shows the percentage of queries in which *Rewriting* and *SMASH* are slower than *Base*. While the rewriting approach yields large improvements in e2e performance, it also causes minor slowdown of queries in over half of all cases. The data illustrates clearly that SMASH provides a convincing solution to the problem, with minor slowdowns on only around 2% of the queries, combined with even larger improvements in e2e performance.

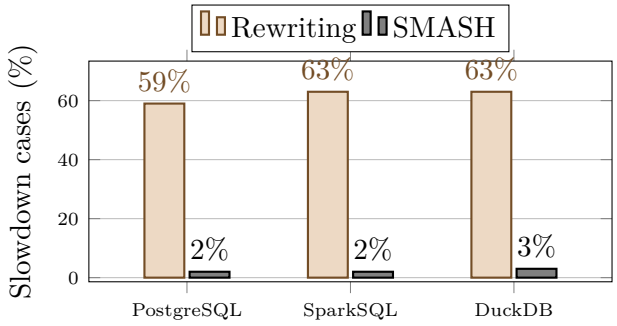


Figure 4.11: Analysis of how often a query is slower than the base case when always rewriting to Yannakakis-style execution vs. when rewriting depending on our trained algorithm selection models (out of 441 queries in the test set).

4.6.5 The Effect of Data Augmentation.

	Base			Augmented		
DBMS	Acc.	Prec.	Rec.	Acc.	Prec.	Rec.
Postgres	0.88	0.83	0.89	0.95	0.95	0.93
DuckDB	0.86	0.79	0.84	0.91	0.88	0.87
SparkSQL	0.85	0.79	0.81	0.93	0.92	0.88

Table 4.11: Ablation study comparing the performance of a model on a training set with augmented data to one without. Values are based on the evaluation of the regression model with a 0.5-threshold on the test set.

We explore the impact of data augmentation on model performance through a focused ablation study, carefully examining the effectiveness of our augmentation strategies. Specifically, we compare models trained on two distinct training sets: one encompassing the fully augmented dataset derived from the complete MEAMBench, and another restricted solely to *base* queries with all augmented data removed. Table 4.11 succinctly summarizes this comparison, presenting accuracy, precision, and recall metrics for both the "Base" set comprised of 0MA and enumeration queries without filter augmentation, and the full augmented training set. Our analysis reveals that incorporating data augmentation substantially enhances accuracy and recall, with an even more pronounced improvement observed in precision.

4.7 Summary

In this chapter, we have studied the effectiveness of Yannakakis-style query evaluation in common, widely used, relational DBMSs on simply structured yet large queries. We observed that these kinds of queries can be highly challenging. On the other hand,

structure-guided query evaluation – executed by the same DBMSs – greatly improves on the number of such queries that are answerable in reasonable time. We introduced the class 0MA – a class of queries which are particularly well suited for structure-guided query processing, and we have shown that Yannakakis-style query evaluation based on query rewriting is feasible. Nevertheless, despite this promising foundation, Yannakakis-style evaluation is often slower than conventional two-way join trees in practice, depending on the specific query and data characteristics. To unlock the full potential of this approach, we framed the choice between Yannakakis-style optimisation and conventional DBMS evaluation as an algorithm selection problem. Our decision procedure delivers substantial performance improvements, across multiple popular RDBMSs, and demonstrates that learning when to apply the optimisation is just as critical as the optimisation itself.

CHAPTER 5

Efficient Join-Aggregate Processing

In Chapter 4 we have implemented a query-rewriting based realisation of Yannakakis' algorithm, and introduced the 0MA class of queries efficiently answerable by completely avoiding materialisation. Now, we will extend the 0MA class to further classes of queries covering most acyclic aggregate queries. To realise Yannakakis-style join processing without any overhead, we will show how to integrate the optimisations into Spark SQL by introducing logical optimisations as well as new physical operators.

We introduce our novel query optimisation techniques on logical query plans, as well as the classes of *guarded* and *piecewise-guarded* aggregate queries, in Section 5.1. The new physical operator *AggJoin* is described in Section 5.2. In Section 5.3, we report on the experimental evaluation of the implementation over the benchmark queries. The *GroupAggJoin* operator for the extension to unguarded queries, as well as the performance evaluation on the new benchmark for unguarded queries, is presented in Section 5.4. We summarise the results of this chapter in Section 5.5.

5.1 Guarded and Piecewise-Guarded Queries

In Chapter 4, we have introduced the definition of 0MA (zero-materialisation answerable) queries. Recall that queries in this class, which have to satisfy the set-safety and guardedness conditions, can be evaluated by rooting the join tree at the node labelled by the guard and then executing the first bottom-up traversal of Yannakakis' algorithm. This means, that all joins are replaced by semi-joins. The grouping and aggregation can then be evaluated by considering only the resulting relation at the root node.

However, the set-safety condition is quite restrictive in that it is only satisfied by a few aggregate functions – primarily `MIN`, `MAX`, and `COUNT DISTINCT`. The vast majority of

aggregate functions – in particular, `COUNT` (without `DISTINCT`), `SUM`, `AVG`, and the entire collection of statistical aggregate functions provided by the ANSI SQL standard are thus disallowed. For instance, the query in Figure 1.2 involving the `MEDIAN` aggregate is not 0MA.

In this section, we significantly extend the class of queries with aggregates on top of join queries that can be evaluated without actually materialising any joins. To this end, we will first drop the set-safety condition in Section 5.1.1 and then also introduce a relaxation of guardedness in Section 5.1.2. To emphasize the smooth integration of our optimisations into standard SQL execution technology, we will describe our optimisations in the form of equivalence-preserving transformations of Relational Algebra sub-expressions, which can be applied anywhere in the logical query plan.

5.1.1 Guarded Aggregate Queries

In order to cover *all* aggregate functions of the ANSI SQL standard, we now drop the set safety condition and define the class of guarded aggregate queries as follows:

Definition 5.1.1. *Let Q be a query of the form given in Equation (2.1), i.e., $Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)](R_1 \bowtie \cdots \bowtie R_n)$. We call Q a* guarded aggregate query *(or simply, "guarded query"), if $(R_1 \bowtie \cdots \bowtie R_n)$ is acyclic and there exists a relation R_i (= the* guard*) that contains all attributes that are either part of the grouping or occur in one of the aggregate expressions. If several relations have this property, we arbitrarily choose one as the guard.*

Note that we consider an aggregate expression `COUNT(*)` as trivially guarded, since it contains no attributes at all. We will now show that, for any aggregate functions of the ANSI SQL standard, guarded queries can be evaluated without propagating any join results up the join tree. To this end, we revisit an extension of Yannakakis' algorithm by Pichler and Skritek [132] to acyclic queries with a `COUNT(*)` aggregate on top. We adapt this approach to integrate it into the logical query plan of relational query processing, and we further extend it to all other aggregate functions.

The crucial idea for evaluating a query Q of the form given in Equation (2.1) is to propagate frequencies up the join tree rather than duplicating tuples. It is convenient to introduce the following notation: let u denote a node in the join tree T and let T_u denote the set of all nodes in the subtree rooted at u. Moreover, for any node u in T, we write $R(u)$ to denote the relation labelling node u and we write $Att(u)$ to denote the list of attributes of $R(u)$. The goal of the bottom-up propagation of frequencies is to compute, for every node u in T, the result of the following query:

$$\gamma[Att(u), \texttt{COUNT(*)}]\left(\bowtie_{v \in T_u} R(v)\right) \tag{5.1}$$

This propagation is realised by recursively constructing extended Relational Algebra expressions $Freq(u)$ for every node u of the join tree, such that $Freq(u)$ gives the same

result as the query in Equation (5.1). Hence, $\mathit{Freq}(u)$ has as attributes all attributes of $R(u)$ plus one additional attribute (which we will denote as c_u), where we store frequency information for each tuple of $R(u)$. If u is a leaf node of the join tree, then we initialise the attribute c_u to 1. Formally, we thus have $\mathit{Freq}(u) = R(u) \times \{(1)\}$.

Now consider an internal node u of the join tree with child nodes $u_1, \ldots, u_k$. The extended Relational Algebra expression $\mathit{Freq}(u)$ is constructed iteratively by defining sub-expressions $\mathit{Freq}_i(u)$ with $i \in \{0, \ldots, k\}$. To avoid confusion, we refer to the frequency attribute of such a sub-expression $\mathit{Freq}_i(u)$ as c_u^i. That is, each relation $\mathit{Freq}_i(u)$ consists of the same attributes $\mathit{Att}(u)$ as $R(u)$ plus the additional frequency attribute c_u^i. Then we define $\mathit{Freq}_i(u)$ for every $i \in \{0, \ldots, k\}$ and, ultimately, $\mathit{Freq}(u)$ as follows:

$$
\begin{aligned}
\mathit{Freq}_0(u) &:= R(u) \times \{(1)\} \\
\mathit{Freq}_i(u) &:= \gamma[\mathit{Att}(u), c_u^i \leftarrow \mathtt{SUM}(c_u^{i-1} \cdot c_{u_i})](\mathit{Freq}_{i-1}(u) \bowtie \mathit{Freq}(u_i)) \\
\mathit{Freq}(u) &:= \rho_{c_u \leftarrow c_u^k}(\mathit{Freq}_k(u))
\end{aligned}
$$

Intuitively, after initialising c_u^0 to 1 in $\mathit{Freq}_0(u)$, the frequency values $c_u^1, \ldots, c_u^k$ are obtained by grouping over the attributes $\mathit{Att}(u)$ of $R(u)$ and computing the number of possible extensions of each tuple $t \in R(u)$ to the relations labelling the nodes in the subtrees rooted at $u_1, \ldots, u_k$. By the connectedness condition of join trees, these extensions are independent of each other, i.e., they share no attributes outside $\mathit{Att}(u)$. Moreover, the frequency attributes $c_u^1, \ldots, c_u^k$ are functionally dependent on the attributes $\mathit{Att}(u)$. Hence, by distributivity, the value of c_u^k obtained by iterated summation and multiplication for a given tuple t of $R(u)$ is equal to computing, for every $i \in \{1, \ldots, k\}$ the sum s_i of the frequencies of all join partners of t in $\mathit{Freq}(u_i)$ and then computing their product, i.e., $c_u = c_u^k = \Pi_{i=1}^k s_i$.

In the logical query plan of query Q, we replace the sub-expression corresponding to the join query $R_1 \bowtie \cdots \bowtie R_n$ by $\mathit{Freq}(r)$, where r is the root node of the join tree. This root node was chosen in such a way that $R(r)$ contains all grouping attributes $g_1, \ldots, g_\ell$. Hence, the grouping can be applied to $\mathit{Freq}(r)$ in the same way as to the original join query. Also, the set-safe aggregates (such as `MIN`, `MAX`, `COUNT DISTINCT`) can be applied to $\mathit{Freq}(r)$ "as usual" by simply ignoring the additional attribute c_r. However, all other (i.e., not set-safe) aggregate functions have to be replaced by variants that take the special frequency attribute c_r into account. We thus modify the aggregate functions in expressions like $\mathtt{COUNT}(*)$, $\mathtt{COUNT}(B)$, $\mathtt{SUM}(B)$, and $\mathtt{AVG}(B)$ so that they directly operate on tuples with frequencies. For instance, let B be an attribute of the guard $R(r)$ (and, hence, also of $\mathit{Freq}(r)$). Then, in SQL-notation, we can rewrite common aggregate expressions as follows:

- $\mathtt{COUNT}(*) \rightarrow \mathtt{SUM}(c_r)$
- $\mathtt{COUNT}(B) \rightarrow \mathtt{SUM}(\mathtt{IF}(\mathtt{ISNULL}(B), 0, c_r))$
- $\mathtt{SUM}(B) \rightarrow \mathtt{SUM}(B \cdot c_r)$

- $\texttt{AVG}(B) \rightarrow \texttt{SUM}(B \cdot c_r)/\texttt{COUNT}(B)$

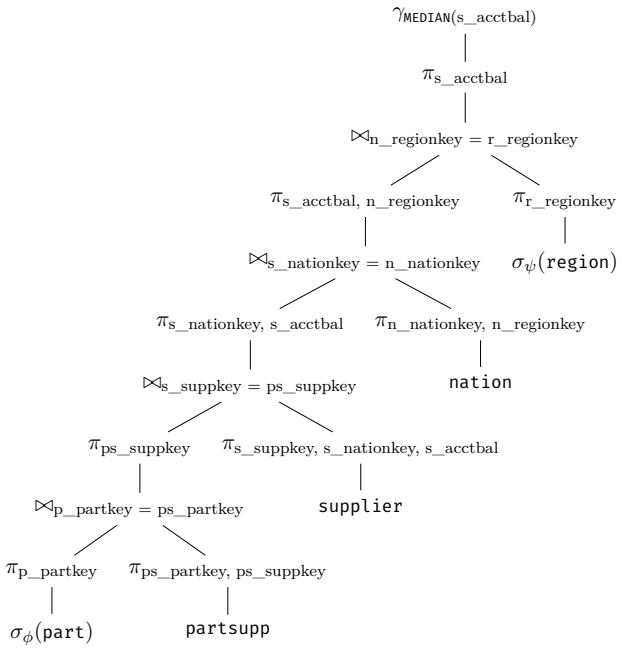
The MEDIAN aggregate (like any other statistical function) can be evaluated by considering $Freq(r)$ as a compressed form of the list of all values of attribute B in each group, where the value of the additional attribute c_r indicates the number of copies of the corresponding value of attribute B in the result of the join query $R_1 \bowtie \cdots \bowtie R_n$. Evaluating an aggregate expression MEDIAN(B) or any other statistical function such as STDDEV(B) can be easily realised for this compressed form of value list. Similarly, the evaluation of aggregate functions on 2 attributes such as CORR(B_1, B_2) or aggregate expressions involving functions on several attributes such as SUM(f(B_1, ..., B_k)) is straightforward by considering $Freq(r)$ as a compressed form of the list of all values of the attribute combinations $B_1, \ldots, B_k$. Again, this crucially depends on the guardedness property, which guarantees that all attributes used in aggregate expressions are contained in $R(r)$.

Actually, in Spark SQL, the MEDIAN aggregate has a convenient rewriting via the PERCENTILE function. The latter is not part of the ANSI SQL standard, but can be found in Spark SQL. This function allows one to provide a frequency attribute, which Spark uses to build a map of values and frequencies, sort them, and finally find the desired percentile value by an efficient search on the sorted map. The rewriting of the MEDIAN aggregate looks as follows:

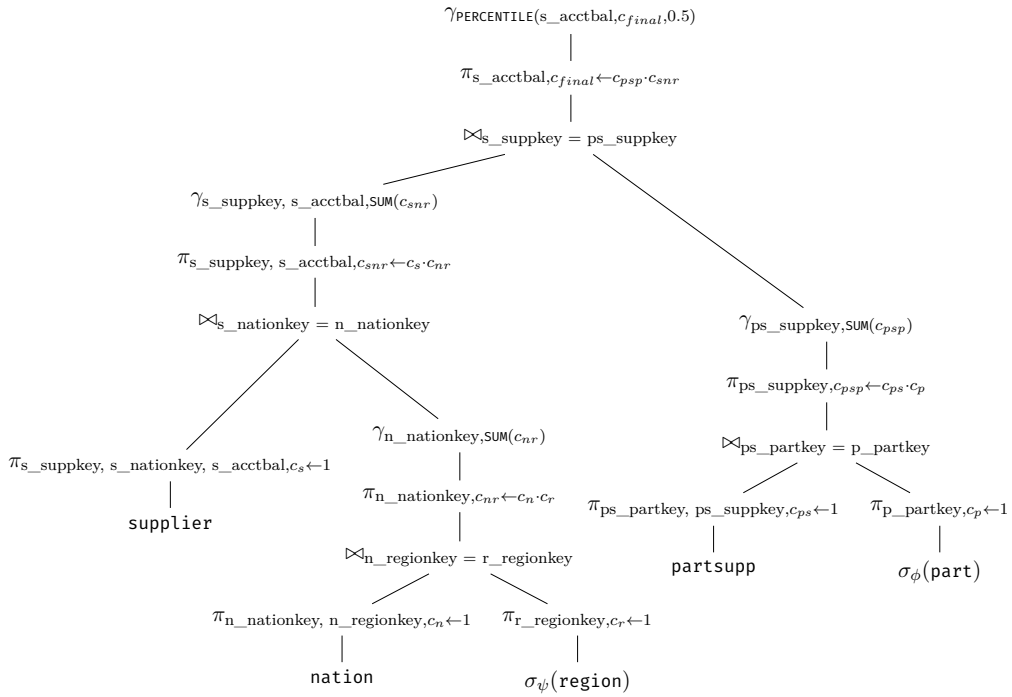
- $\texttt{MEDIAN}(A) \rightarrow \texttt{PERCENTILE}(0.5, A, c_r)$

Example 5.1.1. *Consider again the query of Figure 1.2. The logical query plan generated by Spark SQL is shown in Figure 5.1a. There, we write σ_ψ and σ_ϕ to denote the selections applied to the relations* `region` *and* `part`*, respectively. That is, ψ checks the condition* `r_name IN ('Europe', 'Asia')` *and ϕ checks the condition* `p_price > (SELECT avg (p_price) FROM part)`*. The plan produced by Spark SQL including our optimisation is shown in Figure 5.1b.*

We observe that, in the unoptimised query plan, the entire join of all relations is computed before the `MEDIAN` *aggregate is applied. In contrast, in the optimised plan, only the additional frequency attribute has to be propagated upwards in the plan. This propagation of frequencies for each join is realised by 2 nodes in the plan directly above the node realising the join: first, as part of the projection to the attributes which are used further up in the plan, the frequency attributes of the two join operands are multiplied with each other. Here, we use the notation c_{xy} when frequency attributes c_x and c_y are combined. In the second step, these frequency values c_{xy} are summed up or, in case of the final result, their median is computed, which can be further optimised by making use of the* `PERCENTILE` *function.* ◇



(a) Query plan generated by Spark SQL



(b) Query plan generated by Spark SQL with rewritten aggregation

Figure 5.1: Query plans for Example 5.1.1

We conclude this subsection with an example where we display the information that has to be propagated in the optimised evaluation of the query from Figure 1.2. Actually, it is illustrative to first observe how the tree structure of the join tree is transformed into the tree structure of the optimised plan. Of course, in the latter, the relations must be at the leaf nodes, whereas, in the former, they also occur at inner nodes. Nevertheless, the bushy optimised plan clearly reflects the join order from the join tree. That is, first, `region` and `nation` are joined to get intermediate result-1, and `part` and `part_supp` are joined to get intermediate result-2. The join of these two intermediate results with the relation `supplier` is then split into two 2-way joins, i.e.: first joining `supplier` with result-1, which is then joined with result-2. Hence, for the sake of simplicity, we will discuss the evaluation of this query by looking at the relations at each node of the join tree. It is then clear, what the intermediate results at the nodes of the logical plan in Figure 5.1b look like.

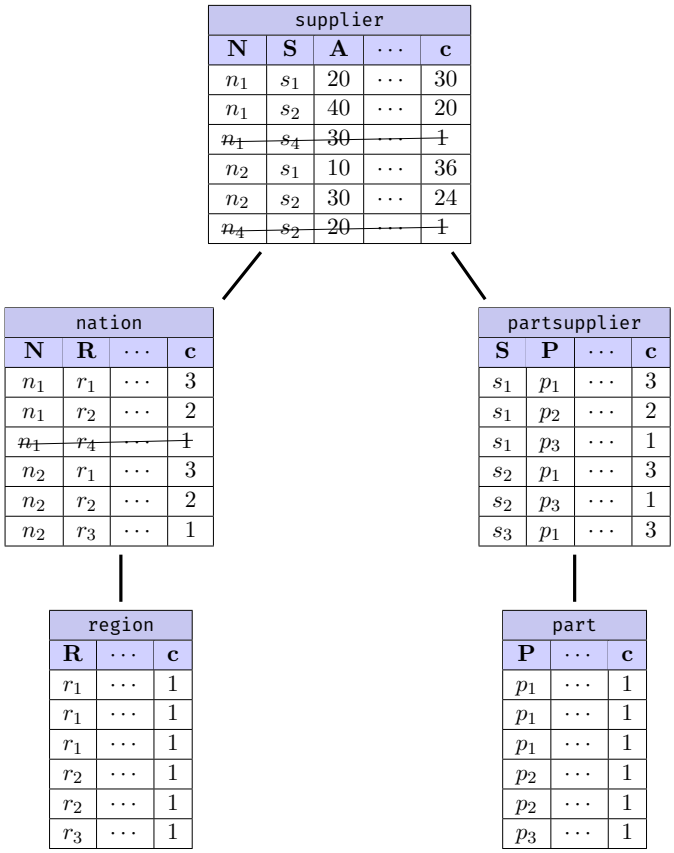


Figure 5.2: Evaluation of the query from Figure 1.2

Example 5.1.2. *Consider again the query from Figure 1.2. Note that all joins in this query are along foreign-key/primary-key relationships. For the sake of illustration, let us ignore the primary keys for a while and allow multiple occurrences of values in these*

attributes. In Figure 5.2, we illustrate the evaluation of the query on a small sample database. The tables (with attribute names of the join attributes abbreviated to single characters) are arranged in the form of the join tree. Attributes not relevant to our discussion are captured by "...". The original contents of the tables is shown to the left of the "..." column. In the right-most column, we display the frequency attribute c for each tuple at the end of the entire bottom-up traversal. For instance, the `Region` *table has 3 tuples with attribute value* $R = r_1$*. Hence, all tuples in* `Nation` *with* $R = r_1$ *have* $c = 3$*, i.e., the number of possible extensions to the subtree below. The tuple with value* $R = r_4$ *is deleted since it has no join partner below. In the root node, the sums of the frequency attributes of all join partners to the left and to the right are multiplied. For instance, the tuple with attribute values* $N = n_1$ *and* $S = s_1$ *has 5 possible extensions to the subtree on the left and 6 on the right. Hence, for this tuple, we get* $c = 30$*.*

For the evaluation of `MEDIAN(A)` *we see at the root node, that the first tuple has attribute value* $A = 20$ *and its frequency in the overall join result is* $c = 30$*. Likewise, the values* $A = 40$*, 10, 30 occur 20, 36, and 24 times, respectively in the join result. We thus get* `MEDIAN(A)` $= 20$*.* ◇

5.1.2 Piecewise-Guarded Aggregate Queries

As we will see in the experimental evaluation in Section 5.3, the class of guarded queries covers significantly more cases from common benchmarks than 0MA. However, the requirement of a single guard for *all* attributes occurring in the `GROUP BY` clause or in any of the aggregate expressions is still quite restrictive. In this section, we show that for the most commonly used aggregate functions `MIN`, `MAX`, `SUM`, `COUNT`, and `AVG`, we can further extend the class of queries that can be evaluated without materialising any joins. We thus introduce the class of piecewise-guarded aggregate queries:

Definition 5.1.2. *Let* Q *be a query of the form given in Equation (2.1), i.e.,* $Q = \gamma[g_1, \ldots, g_\ell,\ A_1(a_1), \ldots, A_m(a_m)](R_1 \bowtie \cdots \bowtie R_n)$*. We call* Q *a* piecewise-guarded aggregate query *(or simply, "piecewise-guarded query"), if* $(R_1 \bowtie \cdots \bowtie R_n)$ *is acyclic and there exists a relation* R_{i_0} *that contains all grouping attributes and, for every* $j \in \{1, \ldots, m\}$*, the following conditions hold:*

- *If* $A_j \in \{$`MIN`, `MAX`, `SUM`, `COUNT`, `AVG`$\}$*, then there exists a relation* R_{i_j} *that contains all attributes occurring in* $A_j(a_j)$*.*
- *Otherwise, i.e.,* $A_j \notin \{$`MIN`, `MAX`, `SUM`, `COUNT`, `AVG`$\}$*, then* R_{i_0} *contains all attributes occurring in* $A_j(a_j)$*.*

Each of these relations R_{i_0} *and* R_{i_j} *is called the "guard" of the corresponding set of attributes. We refer to* R_{i_0} *as the* root *guard. By slight abuse of notation, we also refer to the nodes labelled by* R_{i_0} *and* R_{i_j} *as guards. If several relations could be chosen as guard for a group of attributes, we arbitrarily choose one.*

For the evaluation of piecewise-guarded queries, we choose the node of the join tree T corresponding to the root guard as the root node of T. The bottom-up propagation of the frequency attribute works exactly as for guarded queries. Hence, also the evaluation of all aggregate expressions that are guarded by the root guard is realised exactly as in case of guarded queries. In the rest of this section, we concentrate on the evaluation of aggregate expressions that are *not guarded by the root node* of the join tree and whose aggregate function is one of `MIN`, `MAX`, `SUM`, `COUNT`, and `AVG`. Clearly, the evaluation of `AVG` is based on `SUM` and `COUNT`. Hence, it suffices to describe the evaluation of the remaining four aggregate functions.

Similarly to Koch et al. [94], we extend the relations by additional attributes to carry information on aggregate expressions. Below, we describe which information has to be propagated up the join tree in order to evaluate a single aggregate expression $A_j(a_j)$. For the evaluation of Q, we add the frequency attribute plus *all* these additional attributes to the corresponding nodes in the join tree.

Suppose that $A_j(a_j)$ is of the form $A_j(f_j(\bar{B}_j)$ with $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{COUNT}\}$ and f_j is an arbitrary function on attributes $\bar{B}_j$ jointly occurring in one of the relations $R_1, \ldots, R_n$. We choose as guard of the aggregate expression $A_j(a_j)$ the node w that contains all attributes $\bar{B}_j$ and that is highest up in the join tree T with this property. Since we are assuming that $A_j(a_j)$ is not guarded by the root node r of T, this means that w is different from r. Then we add to all relations along the path from w to r an additional attribute Agg_j. Analogously to Equation (5.1), the intended meaning of Agg_j for every node u on the path between w and r is as follows:

$$\gamma[Att(u), Agg_j \leftarrow A_j(f_j(\bar{B}_j)](\bowtie_{v \in T_u}(R(v))), \tag{5.2}$$

For the *initialisation* of Agg_j, suppose that the frequency attribute of relation $R(w)$ at node w has already been computed as described in Section 5.1.1. Hence, in particular, $R(w)$ is restricted to the tuples t with positive frequency. For an arbitrary tuple t in $R(w)$, we write $t.c$, $t.Agg_j$, and $t.\bar{B}_j$ to denote the values of t at the frequency attribute c, at the aggregate attribute Agg_j, and at the attributes $\bar{B}_j$, respectively. Then we define $t.Agg_j$ as follows:

- If $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$, then we set $t.Agg_j := f_j(t.\bar{B}_j)$.
- If $A_j = \texttt{COUNT}$, then we distinguish two cases: If $f_j(t.\bar{B}_j) = \texttt{NULL}$, then we set $t.Agg_j := 0$; otherwise $t.Agg_j := t.c$.
- If $A_j = \texttt{SUM}$, then we set $t.Agg_j := f_j(t.\bar{B}_j) * t.c$.

To verify that Agg_j is equal to the additional attribute according to Equation (5.2), we note that all tuples in a group defined by a value combination of the original attributes $Att(w)$ of $R(w)$ (thus, corresponding to a single tuple $t \in R(w)$) coincide on the attributes

$\bar{B}_j$. Hence, the MAX and MIN of $f_j(\bar{B}_j)$ over the tuples in such a group is simply the value of $f_j(t.\bar{B}_j)$. For $A_j \in \{\texttt{COUNT}, \texttt{SUM}\}$, we have to take the number of tuples in each such group into account, which corresponds to the frequency value $t.c$. For the COUNT aggregate, we also have to consider the special case that $f_j(\bar{B}_j) = \text{NULL}$, which means that $\texttt{COUNT}(f_j(\bar{B}_j))$ for the entire group is 0.

For the *propagation* of the additional attribute Agg_j along the path from w to the root r, consider an ancestor node u of w and let $u_1, \ldots, u_k$ denote the child nodes of u. W.l.o.g., we assume that the child node u_1 is on the path from w to r. Suppose that the frequency attribute at every child node u_i of u and the attribute Agg_j at node u_1 have already been computed. We are assuming that w is the highest node in the join tree that contains all attributes $\bar{B}_j$. Hence, u does not contain all attributes $\bar{B}_j$, and, by the connectedness conditions, neither does any of the nodes $u_2, \ldots, u_k$. For an arbitrary tuple $t \in R(u)$, let $\{t_1, \ldots, t_\alpha\}$ denote the set of all tuples in $R(u_1)$ that join with t. We compute the value $t.Agg_j$ as follows:

- First suppose that $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$. Then we set $t.Agg_j := A_j(\{t_1.Agg_j, \ldots, t_\alpha.Agg_j\})$.
- Now let $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$. For every $i \in \{2, \ldots, k\}$, let s_i denote the sum of the frequencies of all join partners of t in $R(u_i)$. Then we set $t.Agg_j := \left(\sum_{\lambda=1}^{\alpha} t_\lambda.Agg_j[u_1]\right) * \Pi_{i=2}^{k} s_i$.

For the correctness of this propagation of attribute Agg_j, recall that we are assuming that the attributes $\bar{B}_j$ are not fully contained in the relation $R(u)$ and, hence, by the connectedness condition, they cannot be fully contained in any of the child nodes $\{u_2, \ldots, u_k\}$ either. Hence, the value combinations of $\bar{B}_j$ in $(\bowtie_{v \in T_u}(R(v))$ must already occur in $(\bowtie_{v \in T_{u_1}}(R(v))$. The MIN or MAX of $f_j(\bar{B}_j)$ of a tuple t when grouping over the attributes of $R(u)$ is, therefore, simply obtained by grouping over the attributes of $R(u_1)$ and aggregating over the join partners of t in $R(u_1)$. Similar considerations apply to the computation of Agg_j in case of COUNT and SUM. The aggregation of Agg_j over the join partners of t in u_1 yields the value $\left(\sum_{\lambda=1}^{\alpha} t_\lambda.Agg_j\right)$. In contrast to MIN and MAX, we now also have to take the possible extensions of t to the relations in the subtrees of T rooted at the nodes $u_2, \ldots, u_k$ into account. The number of possible extensions of t to $(\bowtie_{v \in T_{u_i}}(R(v))$ corresponds to the sum s_i of the frequencies of all join partners of t in $R(u_i)$. Hence, by the connectedness conditions, the number of extensions of t to the relations at *all* subtrees of $T_{u_2}, \ldots, T_{u_k}$ is obtained as $\Pi_{i=2}^{k} s_i$.

We make an important observation concerning the *size of the relations* that we propagate up the join tree T: for every node u of T, the relation $Freq(u)$ contains precisely the tuples of $R(u)$ that one would get by the first bottom-up traversal of Yannakakis' algorithm via semi-joins, extended by the frequency attribute c_u. That is, we never add tuples, we only add one attribute to each relation. Similarly, for every aggregate expression $A_j(f_j(\bar{B}_j))$ that is not guarded by the root r of T, we add an attribute Agg_j to all nodes

along the path between the guard of $A_j(f_j(\bar{B}_j))$ and the root. In other words, the data structures that we have to materialise and propagate in the course of our evaluation of piecewise-guarded queries is *linearly bounded* in the size of the data.

This property is no longer guaranteed for one of the following two extensions of the piecewise-guarded fragment: either allowing aggregates other than `MIN`, `MAX`, `SUM`, `COUNT`, `AVG` to be guarded by a relation different from the root guard or allowing aggregate expressions $A_j(f_j(\bar{B}_j))$ whose attributes are not guarded by a single relation. In both cases, one has to propagate (all possible) individual values of attributes rather than aggregated values up the join tree, which would destroy this linear bound. In Section 5.4, we will present extensions for going beyond piecewise-guarded queries, which allows us to cover such queries as well.

We now illustrate the evaluation of piecewise-guarded aggregate queries by extending Example 5.1.2.

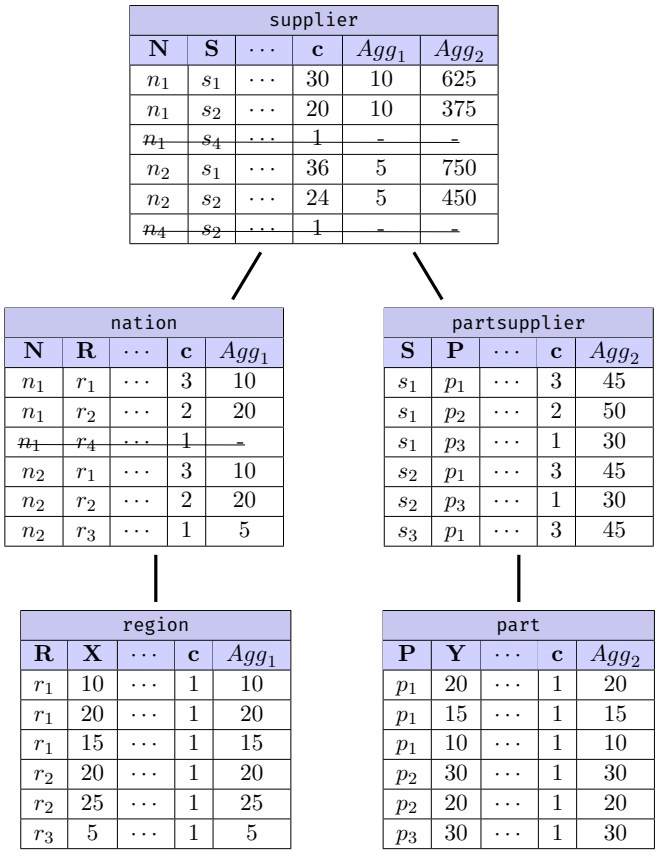


Figure 5.3: Evaluation of the query from Example 5.1.3

Example 5.1.3. *Consider again the query from Figure 1.2. But now we assume that relation* `Region` *has an additional integer attribute* X *and relation* `Part` *has an additional*

*integer attribute Y. Finally, suppose that we want to evaluate the query "*SELECT MIN(X), SUM(Y) ... GROUP BY N*". The query is trivially piecewise-guarded, since we have single attributes in the* `GROUP BY` *clause and in each aggregate expression. On the other hand, it is not guarded, since the attributes X, Y, N do not jointly occur in a common relation.*

In Figure 5.2, we illustrate the evaluation of the query on a small sample database. The tables (with attribute names of the join attributes abbreviated to single characters) are arranged in the form of the join tree. Note that we again choose the `Supplier` *relation as root guard. Alternatively, we could have chosen the* `Nation` *relation, since it also contains the grouping attribute N.*

As in Figure 5.2, attributes not relevant to our discussion are captured by "...". The original contents of the tables is shown to the left of the "..." column. To the right of the "..." column, we now have the frequency attribute c plus the aggregate attributes Agg_1 and Agg_2 for the evaluation of the `MIN` *(X) and* `SUM` *(Y) aggregates as described in Section 5.2. The Agg_1 attribute (for the* `MIN` *(X) aggregate expression) is added to all relations along the path from the node with relation* `Region` *up to the root node, while the Agg_2 attribute (for the* `SUM` *(Y) aggregate expression) is added to all relations along the path from the node with relation* `Part` *up to the root node.*

The frequency attribute c is propagated up the join tree exactly as in Example 5.1.2. For the aggregate attributes $Agg1$ and Agg_2, we proceed as follows: At the leaf nodes (in this simple example, these are the highest nodes up in the tree that contain the attributes X and Y, respectively), we initialise, Agg_1 in every tuple to the value of X and Agg_2 to the value of Y. The value of Agg_1 is propagated from the `Region` *relation to its parent node by taking, for every tuple in* `Nation`*, the* `MIN` *over all its join partners in* `Region`*. Analogously, the value of Agg_2 is propagated to the parent node by taking, for every tuple in* `Partsupplier`*, the* `SUM` *over all its join partners in* `Part`*.*

Now let us look at relation `Supplier` *at the root node. The attribute Agg_1 is propagated from* `Nation` *to* `Supplier` *by taking, for every tuple at the parent node, the* `MIN` *over the Agg_1 values of its join partners at the child node. For instance, the* `Supplier`*-tuple with attribute values $N = n_1$ and $S = s_1$ has two join partners in the* `Nation` *table; the* `MIN`*-value of their Agg_1-attributes is 10. The same holds for the tuple with $N = n_1$ and $S = s_2$. On the other hand, the tuples in* `Supplier` *with $N = n_2$ have the* `MIN`*-value 5 in the Agg_1-attribute of their join partners. As was described in Section* 5.1.2*, the other child node of the (node labelled by the)* `Supplier` *relation plays no role when propagating an aggregate attribute for a* `MIN` *or* `MAX` *aggregate expression.*

This is in sharp contrast to the Agg_2 attribute, where we propagate, for every tuple in the root node, the `SUM` *over the Agg_2 values of the join partners from the right child node. But then we also have to multiply this value by the sum over the frequency-values of all join partners in the left child node. For instance, take the tuple with attribute values $N = n_1$ and $S = s_1$ in* `Supplier`*. On the one hand, the sum over the Agg_2-attributes of its join partners in* `Partsupplier` *(i.e., the tuples with attribute value $S = s_1$) is 125. But then we have to multiply this value by the sum over the c-attributes of its join partners in*

`Nation` *(i.e., the tuples with attribute value* $N = n_1$*), which is 5. Hence, the* Agg_2*-value of the* `Supplier`*-tuple is 125 * 5 = 625. Analogously, the tuples in the* `Supplier` *relation with attribute values* ($N = n_1$ *and* $S = s_2$*), (*$N = n_2$ *and* $S = s_1$*), and (*$N = n_2$ *and* $S = s_2$*) are obtained as 75*5 = 375, 125*6 = 750, and 745*6 = 450, respectively.*

For the evaluation of `MIN(X)` *and* `SUM(Y)` *for each* N*-group, we then just need to aggregate the* Agg_1 *and* Agg_2 *attributes for each group. That is, we have two groups* n_1 *and* n_2*. There, respective values of* (`MIN(X)`, `SUM(Y)`) *are* $(10, 1000)$ *and* $(5, 1200)$*, respectively.* ◇

5.1.3 Additional Optimisations for FK/PK-Relatonships

Joins are frequently performed along foreign-key/primary-key (FK/PK) relationships. Knowledge about these relationships may actually allow us to replace joins in the frequency-propagation optimisation by semi-joins when the joins go along an FK/PK relationship such that the relation labelling the parent node in the join tree holds the FK and the relation at a child node holds the PK. This is due to the fact that, in this case, we know that every tuple of the relation at the parent node can have at most one join partner in the relation at the child node.

In particular, suppose that in the relation at the child node, all tuples have frequency 1. This is guaranteed if the child is a leaf node in the join tree. Then the frequency propagation from the child node to the parent node comes down to a semi-join. That is, we have to check for each tuple in the relation at the parent node if it has a join partner in the relation at the child node (which essentially means that it has no NULL value in an FK-attribute). If so, the parent inherits the frequency 1 from the child node; otherwise, we may simply discard this tuple from the relation at the parent node. Of course, then the same consideration may be iterated also for the join with the parent of the parent, if this is again along an FK/PK-relationship. We illustrate this additional optimisation by revisiting Example 5.1.1.

Example 5.1.4. *Consider again the query from Figure 1.2. An inspection of the join tree in Figure 1.2 and of the TPC-H schema reveals that all joins are along FK/PK-relationships from the relation at the parent node to the relation at the child node. We can, therefore, be sure that all frequency attributes in our optimisation can only take the value 1. Hence, all joins can be replaced by semi-joins. The logical query plan produced by Spark SQL when implementing this additional optimisation is shown in Figure 5.4. Here, for each FK/PK-relationship, the referencing relation is shown as the left child of the* $\ltimes$*-node in the query plan and the referenced relation as the right child of the* $\ltimes$*-node. The semi-join is always from the referenced relation into the referencing relation, i.e., from right to left.* ◇

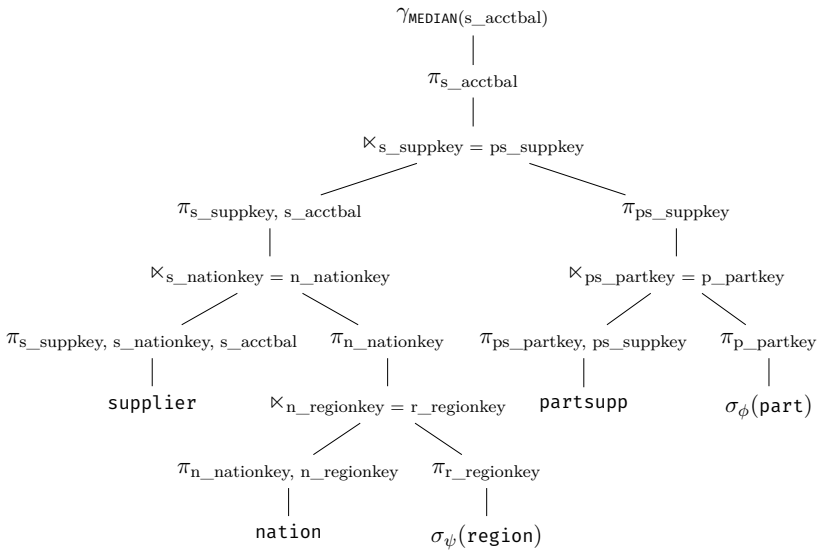


Figure 5.4: Query plan generated by Spark SQL with FK/PK optimisation

The information on primary keys or, more generally, on unique attributes can be exploited for a further optimisation: Recall from Example 5.1.1 and Figure 5.1b that, in the optimised query plan, the multiplication of the frequency c_x at the parent with the frequency c_y at the child is followed by a projection (actually, a grouping) and a summation of the c_{xy} values. Now suppose that the grouping attributes (i.e., the attributes to which we project) contain an attribute or a set of attributes with unique values. Then, each value combination of the grouping attributes occurs at most once. Hence, the summation of the c_{xy} values can be omitted in this case.

Finally, the information on primary keys or unique attributes can be used for yet another optimisation: as was explained in Section 5.1, our optimisation of guarded aggregates starts with adding frequency 1 as an additional attribute to each tuple in the relation R labelling a node u in the join tree, i.e., the computation of the optimised Relational Algebra expression $\mathit{Freq}(u)$ starts with the initialisation $\mathit{Freq}_0(u) := R \times \{(1)\}$. Before joining this relation $\mathit{Freq}_0(u)$ with the relations at the child nodes, we can group the relation at each child node over the attribute(s) relevant further up in the query plan and sum up the frequency values for each group. In cases where value combinations of the grouping attributes occur frequently in such a relation, this significantly reduces the size of one join operand and, hence, the cost of the join. But, of course, such an additional grouping operation would cause useless effort if the grouping attributes contain a PK or unique attribute since, in this case, each group would consist of a single tuple. In our optimisation, we therefore apply this grouping only if no attribute with unique value is involved in the grouping.

5.2 Optimised Physical Operators

The optimisations presented in Section 5.1 avoid a good deal of materialisation of intermediate results. But there are still joins needed, namely, between the relations at a parent node and its child nodes. Only after these joins, we apply the grouping and aggregation and thus bring the intermediate relations back to linear size (data complexity). Similarly to Schleich et al. [139], we now combine the join computation with aggregation into a single operation. We thus introduce a new physical operator (referred to as AggJoin) that computes and propagates the frequency attribute c from Section 5.1.1 and the additional aggregate attributes Agg_j from Section 5.1.2 in a semi-join-like style. Below, we describe a possible *join-less* realisation of this operator. As a preprocessing step (yet before the AggJoin is called), the following operations are carried out:

Every relation is extended by a *frequency attribute* c, and, for every tuple t in every relation R, we initialise $t.c$ as $t.c := 1$. Moreover, for every aggregate expression $A_j(f_j(\bar{B}_j))$ that is not guarded by the root guard, we determine the node w highest up in the join tree that contains all attributes in $f_j(\bar{B}_j)$. Then we add an attribute Agg_j to every relation along the path from w to r. For every tuple $t \in R(w)$, we initialise this attribute as follows:

- If $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}\}$, then we set $t.Agg_j := f_j(t.\bar{B}_j)$, i.e., we apply f_j to the values of the attributes $\bar{B}_j$ in tuple t.
- If $A_j \in \{\texttt{COUNT}\}$, then we set $t.Agg_j := 1$ if $f_j(t.\bar{B}_j) \neq \texttt{NULL}$, and $t.Agg_j := 0$ if $f_j(t.\bar{B}_j) = \texttt{NULL}$.

For $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we thus deviate from the initialisation of $t.Agg_j$ at node w described in Section 5.1.2 by leaving out the multiplication of $t.Agg_j$ with the frequency value $t.c$. This multiplication by $t.c$ has to be integrated into the AggJoin operator, which will take care of this multiplication when it determines $t.c$.

Finally, if $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, then we set $t.Agg_j := 1$ for every tuple t in a relation labelling an ancestor node u of w. The reason for this is that it will allow us to uniformly propagate the Agg_j value from one child of u and the frequency values from the other children of u in a uniform way via multiplication.

From now on, let R and S denote relations labelling nodes u_R and u_S in the join tree, such that u_S is a child node of u_R. We first describe the propagation of the frequency attribute by the AggJoin for a tuple $r \in R$.

- check that $r \in R \ltimes S$ holds;
- define $S' := S \ltimes \{r\}$, i.e., the tuples in S that join with r;
- define $sc := \sum_{s \in S'} s.c$, i.e., the sum of the frequencies of all tuples in S that join with r;

- Finally, we set $r.c := r.c \cdot sc$, i.e., the frequency of r is multiplied by the sum of the frequencies of the join partners in S. Here, it makes no difference, if $r.c$ still had its initial value 1 or if R had already gone through calls of AggJoin with relations at other child nodes of u_R.

It is easy to verify that this new AggJoin operator does precisely the work needed to get from $Freq_{i-1}(u_R)$ to $Freq_i(u_R)$ according to Section 5.1.1. After the initialisation, we have $r.c = 1$ for all tuples in R. This corresponds to $Freq_0(u_R)$. Then we successively execute the AggJoin operator, where $R = Freq_{i-1}(u_R)$ and S is the relation at the i-th child node of u_R. Hence, in each such call, we either delete r (if it has no join partner in S) or we multiply the current value of $r.c$ by the sum of the frequencies of its join partners in S.

Let us now consider the aggregate expressions $A_j(f_j(\bar{B}_j))$ that are not guarded by the root guard. For the *initialisation* of the attribute Agg_j in case of $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we proceed as with the frequency attribute: Suppose that relation R is the one, where Agg_j has to be initialised. Now, for every tuple $r \in R$, the value initially assigned to $r.Agg_j$ ultimately has to be multiplied by $r.c$ to arrive at the initialisation according to Section 5.1.2. Hence, for every relation S at a child node of u_R, we multiply $r.Agg_j$ with $sc =$ the sum of the frequencies of all tuples in S that join with r.

For the *propagation* of the attribute Agg_j in case of $A_j \in \{\texttt{MIN}, \texttt{MAX}, \texttt{SUM}, \texttt{COUNT}\}$, we distinguish two cases: First suppose that S does not contain the attribute Agg_j. Then, for every tuple $r \in R$ that has at least one join partner in S, we proceed as follows: For $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$, we simply leave the value of $r.Agg_j$ unchanged, i.e., the Agg_j attribute is propagated to R from the relation at a different child node. For $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we again proceed analogously to the frequency propagation, i.e., we multiply $r.Agg_j$ with the sum of the frequencies of all tuples in S that join with r. Now suppose that S contains the attribute Agg_j. Then, for every tuple $r \in R$ that has at least one join partner in S, we proceed as follows: For $A_j \in \{\texttt{MIN}, \texttt{MAX}\}$, we assign to $r.Agg_j$ the minimum resp. maximum value of $s.Agg_j$ over all tuples $s \in S$ that join with r. For $A_j \in \{\texttt{SUM}, \texttt{COUNT}\}$, we determine the sum of the values $s.Agg_j$ over all tuples $s \in S$ that join with r and multiply the current value of $r.Agg_j$ with this sum.

The rewriting from Section 5.1 allows for a smooth integration of the AggJoin operator into the physical query plan. For instance, we have extended Spark SQL by three different implementations of the AggJoin operator, corresponding to the existing three join implementations *shuffled-hash join*, *sort-merge join*, and *broadcast-hash join*. In Algorithm 5.1, we sketch the realisation of the AggJoin operator based on the shuffled-hash join. We use pseudocode notation to leave out the technical details so as not to obscure the simplicity of the extension from join computation to semi-join-like aggregation. As in the explanations above, we write R and S to denote pairs of relations whose nodes in the join tree are in parent-child relationship. Moreover, the AggJoin operator is only called after all the initialisations of additional attributes $t.c$ and $t.Agg_j$ have been carried out

Algorithm 5.1: Hash Join with aggregate propagation

Input: Two lists R, S of tuples with the same values of the join attributes;
List $I_S = \{s_1, \ldots, s_m\}$ of indices of aggregate attributes Agg_{s_i} present in both S and R;
List $I_R = \{r_1, \ldots, r_n\}$ of indices of aggregate attributes Agg_{r_i} present only in R;

```
1  Function AggHashJoin(R, S, I_S, I_R)
2      sc ← 0;
3      foreach s ∈ I_S do
4          if A_s ∈ {MIN, MAX} then val_s ← init[s];
5          if A_s ∈ {SUM, COUNT} then val_s ← 0;
6      end
7      foreach t ∈ S do
8          sc ← sc + t.c;
9          foreach s ∈ I_S do
10             if A_s ∈ {MIN, MAX} then val_s ← A_s(val_s, t.Agg_s);
11             if A_s ∈ {SUM, COUNT} then val_s ← val_s + t.Agg_s;
12         end
13     end
14     foreach t ∈ R do
15         t.c ← t.c · sc;
16         foreach s ∈ I_S do
17             if A_s ∈ {MIN, MAX} then t.Agg_s ← val_s;
18             if A_s ∈ {SUM, COUNT} then t.Agg_s ← t.Agg_s · val_s;
19         end
20         foreach r ∈ I_R do
21             if A_r ∈ {SUM, COUNT} then t.Agg_r ← t.Agg_r · sc;
22         end
23         emit t;
24     end
```

as described above. Clearly, the hash-phase (including the partitioning by Spark SQL) is left unchanged. Only the join-phase is affected, which we briefly discuss next:

The AggHashJoin takes as input a set of tuples from R and of tuples from S that join. Additionally, the indices of the aggregate attributes I_S (which have to be propagated from S to R) and I_R (which are only contained in R) are taken as input. In the first step, we initialise sc (that is used to sum up the frequency values over the tuples in S) and val_s for every $s \in I_s$ (that is used for aggregating the attribute Agg_s). An aggregate attribute Agg_s is used to propagate values for the aggregate expression $A_s(f_s(\bar{B}_s))$. For $A_s \in \{\texttt{MIN}, \texttt{MAX}\}$ we assume that the (system-dependent) *maximal* element for this data type in case of `MIN` and the *minimal* one in case of `MAX`, respectively, is stored in the

variable $init[s]$. The foreach-loop over the tuples of S aggregates the frequency attribute and all the other additional attributes. The foreach-loop over the tuples of R uses these aggregated values from the tuples of S to update the corresponding attributes of the tuples in R. The latter foreach-loop also has to multiply the initial value of aggregate attributes in case of `SUM` and `COUNT` by the sum sc of the frequency attributes.

Clearly, replacing a physical join operation by the respective AggJoin variant does not introduce any overhead (apart from the computationally cheap management of the additional frequency and aggregate attributes). Moreover, if none of the additional attributes is needed (e.g., if the query is 0MA), then our AggJoin operator actually degenerates to a simple semi-join.

We now also describe the realisation of the AggJoin operator based on Spark's sort-merge-join. As with the AggHashJoin, we write R and S to denote pairs of relations whose nodes in the join tree are in parent-child relationship. Moreover, we again assume that the AggJoin operator is only called after all the initialisations of the additional attributes $t.c$ and $t.Agg_j$ have been carried out for every tuple $t \in R$ as described in Section 5.2. Clearly, the sort phase is left unchanged. Only the merge-phase is affected, which we briefly discuss next:

In Algorithm 5.2, we sketch the realisation of the AggJoin operator in the style of a merge join. Again, we use pseudocode notation to leave out the technical details. Here, we assume that relations R and S are joined over a common attribute A. The generalization to attributes with different names or to a join over a compound attribute is immediate. Similarly to the AggHashJoin, also the AggMergeJoin has four input parameters: the sorted (on the join attribute A in ascending order) relations R and S as well as the set of indices of the aggregate attributes I_S (which have to be propagated from S to R) and the set of indices of the aggregate attributes I_R (which are only contained in R).

The outer while-loop makes sure that we process all tuples of R and of S. Inside this while loop, we first have a repeat–loop to find the first resp. next pair of join partners (i.e., the minimal resp. next indices i and j with $R[i].A = S[j].A$). Next comes the initialisation of sc and val_s for every $s \in I_S$ exactly as for the AggHashJoin. The while-loop with the condition $R[i].A = S[j].A$ iterates through all join partners of $R[i]$ in S and computes sc and val_s for every $s \in I_S$ exactly as in case of the AggHashJoin. In the do-while-loop, we use these local variables sc and val_s to update the frequency attribute c and all aggregate attributes Agg_j with $j \in I_S \cup I_R$ for all tuples $R[i']$ of R with $i' \geq i$ and $R[i].A = R[i'].A$, i.e., all tuples in R that coincide on A with $R[i]$. Again, this is done exactly as in case of the AggHashJoin. That is, for every such tuple $R[i']$, we first update the frequency attribute $R[i'].c$ and the aggregate attributes $R[i'].Agg_s$ for every $s \in I_S$ by making use of the local variables sc and val_s. We then also update all aggregate attributes Agg_r (with aggregate function `SUM` or `COUNT`) for all attributes $r \in I_R$. When the frequency attribute c and all aggregate attributes of a tuple $R[i']$ have thus been updated, the tuple $R[i']$ is emitted.

Algorithm 5.2: Sort-Merge Join with aggregate propagation

Input: Two lists R, S of tuples with the same values of the join attributes;
List $I_S = \{s_1, \ldots, s_m\}$ of indices of aggregate attributes Agg_{s_i} present in both S and R;
List $I_R = \{r_1, \ldots, r_n\}$ of indices of aggregate attributes Agg_{r_i} present only in R;

```
1  Function AggMergeJoin(R, S, I_S, I_R)
2      i, j ← 1, 1;
3      while i ≤ |R| and j ≤ |S| do
4          repeat
5              while i < |R| and R[i].A < S[j].A do
6                  i ← i + 1;
7              end
8              if R[i].A < S[j].A then return;
9              while j < |S| and R[i].A > S[j].A do
10                 j ← j + 1
11             end
12             if R[i].A > S[j].A then return;
13         until R[i].A = S[j].A;
14         sc ← 0;
15         foreach s ∈ I_S do
16             if A_s ∈ {MIN, MAX} then val_s ← init[s];
17             if A_s ∈ {SUM, COUNT} then val_s ← 0;
18         end
19         while j ≤ |S| and R[i].A = S[j].A do
20             sc ← sc + S[j].c;
21             foreach s ∈ I_S do
22                 if A_s ∈ {MIN, MAX} then val_s ← A_s(val_s, S[j].Agg_s);
23                 if A_s ∈ {SUM, COUNT} then val_s ← val_s + S[j].Agg_s;
24             end
25             j ← j + 1
26         end
27         do
28             R[i].c ← sc · R[i].c ;
29             foreach s ∈ I_S do
30                 if A_s ∈ {MIN, MAX} then R[i].Agg_s ← A_s(val_s, R[i].Agg_s);
31                 if A_s ∈ {SUM, COUNT} then R[i].Agg_s ← R[i].Agg_s · val_s;
32             end
33             foreach r ∈ I_R do
34                 if A_s ∈ {SUM, COUNT} then R[i].Agg_r ← R[i].Agg_r · sc;
35             end
36             emit R[i];
37             i ← i + 1;
38         while i ≤ |R| and R[i].A = R[i − 1].A;
39     end
```

The broadcast-hash join in Spark SQL sends one of the relations to every node that contains a split of the other relation, and determines the matching tuples on each node separately. This last step is realised by a hash join. We have implemented our optimisation by choosing S as the relation that is sent to all nodes where a split of R is located. Then the AggJoin operator is realised by executing the AggHashJoin between the local split of R and the entire relation S.

Of course, it is equally straightforward to implement the AggJoin operator for join types not supported by Spark SQL, such as the block-nested-loops join. In this case, when considering R as the outer relation and S as the inner relation, the frequency attribute c and the aggregate attributes Agg_j of each tuple r in the current block of R can be updated as follows: we provisionally add an sc attribute and val_s attributes for every $s \in I_S$ to every tuple r of the current block of R. Then relation S is traversed and, for every tuple $s \in S$, the sc and val_s attributes of each tuple in the current block of R that joins with s is updated analogously to the while-loop with the condition $R[i].A = S[j].A$ in case of the sort-merge-join. When all of S has been processed, then the frequency attribute and all aggregate attributes Agg_j of every tuple in the current block of R are updated analogously to the do-while-loop in case of the sort-merge-join. If the sc attribute of a tuple in $r \in R$ is 0 (i.e., r has no join partner in S), then we simply delete this tuple of R.

5.3 Experimental Evaluation

5.3.1 Experimental Setup

We perform the experiments on a machine with two AMD EPYC 75F3 32-Core CPUs and 960 GB RAM. On top of this, we use a VM with 60 cores running Ubuntu 22.04.2 LTS and Spark SQL (Version 3.5.0). For our experiments, we implement the optimisations presented in Section 5.1 and the physical operator from Section 5.2 natively in Spark SQL (see Figure 5.5 for an overview of the modified components). Our experimental setup is reproducible through a docker-compose environment available at `https://github.com/dbai-tuw/spark-eval`.

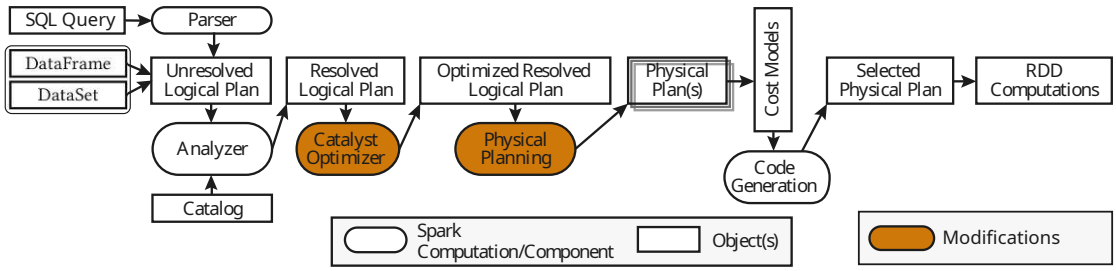


Figure 5.5: Implementation of the optimisations into the Spark SQL query processing pipeline (adapted from [70])

In order to import the benchmark databases into Spark SQL, we run a second container with PostgreSQL 16, from where the data is fetched over JDBC. We configure Spark with 900 GB maximum executor memory and 60 available cores. Off-heap storage (to disk) is disabled in order to avoid performance degradation caused by unexpected use of disk storage for intermediate results. In our experiments, we encountered some instances where the excessive memory consumption caused the query to fail. However, these cases exhibited such extreme performance differences that they do not impact the conclusions drawn from our experimental evaluation. Specifically, queries where Spark SQL failed due to requiring more than 900GB of memory are easily solved by our implementation, using only a small fraction of the available resources. We perform experiments using a wide range of standard benchmarks, namely the *Join Order Benchmark (JOB)* [101], *STATS-CEB* [76], TPC-H [150], TPC-DS [149], the *Large-Scale Subgraph Query Benchmark (LSQB)* [116], as well as simple graph queries evaluated on two real-world graphs from the *SNAP (Stanford Network Analysis Project* [102]) dataset:

- The *Join Order Benchmark (JOB)* [101] is a benchmark based on the IMDB real-world data set and a large number of realistic join-aggregation queries featuring many joins and various filtering conditions. It was introduced to study the join ordering problem and evaluate the performance of query optimisers.
- The *STATS / STATS-CEB* [76] benchmark serves a similar purpose as the JOB benchmark but with the explicit addition of joins that do not follow FK/PK relationships (but rather FK/FK to keep with typical usage patterns). The data is based on anonymized content from Stack Exchange.
- *TPC-H* [150] and TPC-DS [149] are standard benchmarks for relational databases that cover a wide range of complex real-world workloads. In addition, we report performance of our running example query from Example 5.1.1 under the name "TPC-H Ex.1". We use scale factor 200 in the generation of the TPC-H data and scale factor 100 for TPC-DS.
- The *Large-Scale Subgraph Query Benchmark (LSQB)* [116] is a benchmark of 9 graph queries, designed to test the query optimiser and executor of graph databases as well as relational databases. Its schema represents a social network scenario with relations for, e.g., persons, posts, and comments. We generate data with scale factor 300 for LSQB. Data generation is based on the LDBC [10] benchmark, a synthetic benchmark which takes care to closely mirror the properties of real-life social networks. The benchmark consists of 11 relations with various scale factors (e.g., 10, 30, 100, 300), as well as 9 queries in various query languages (SQL, Cypher, etc.).
- The *SNAP (Stanford Network Analysis Project* [102]) dataset is commonly used to benchmark queries on graph data (e.g., [82]). Most SNAP datasets consist of a single, directed or undirected, edge relation, such as the US patent citation dataset

(cit-Patents), or the Google web graph (web-Google). In particular, we experiment on the following two popular graphs of various sizes:

Graph	Nodes	Edges	(un)directed
web-Google	875,713	5,105,039	directed
com-DBLP	317,080	1,049,866	undirected

For our experiments, we evaluate the performance of basic graph queries, namely path queries requiring between 3 and 8 joins (i.e., between 4 and 9 edges) and three small tree queries. For example, the path with 3 joins (path-03) over the web-Google graph is expressed in SQL as

```
SELECT COUNT(*) FROM
edge e1, edge e2, edge e3, edge e4
WHERE e1.toNode = e2.fromNode
 AND  e2.toNode = e3.fromNode
 AND  e3.toNode = e4.fromNode
```

The tree queries can be found in the GitHub repository[1]. tree-01 has a depth between 3 and 4 in total, tree-02 has depth 3 and 5 joins, and tree-03 has depth 4 and 7 joins. These queries can be viewed as counting the number of homomorphisms from certain patterns (i.e., paths and trees in this case). This task has recently gained popularity in graph machine learning, where the results of the queries are injected into machine learning models (e.g., [123, 18, 87, 16]).

Benchmark	#	⋈-agg	acyc	pwg	g	0MA
JOB	113	113	113	113	19	19
STATS-CEB	146	146	146	146	146	0
TPC-H	22	15	14	7	3	1
LSQB	9	4	2	2	2	0
SNAP	18	18	18	18	18	0
TPC-DS	99	64	63	30	15	0

Table 5.1: Summary of the applicability of our method on benchmarks. We report the number of queries in benchmark (#), equi-join aggregate queries (⋈-*agg*), acyclic queries (*acyc*), piecewise-guarded queries (*pwg*), guarded queries (*g*), and 0MA queries. Fragments proposed in this work are highlighted in blue.

5.3.2 Applicability

To enable Yannakakis-style query evaluation in the context of standard query execution engines, we have focused on specific queries, namely guarded and piecewise-guarded acyclic aggregate queries (cf., Section 5.1). As a first step, we therefore provide a more

[1] https://github.com/dbai-tuw/spark-eval/blob/main/benchmark/snap-queries/google

detailed analysis as to how many of the queries in the studied benchmarks fit into this class, and what factors limit further applicability. The analysis is summarized in Table 5.1.

Despite the variety of considered benchmarks, we find that our optimisations for piecewise-guarded queries are widely applicable through all of them. In JOB and STATS-CEB, all queries fall into the schema of piecewise guarded-aggregation, and our method thus applies to these benchmarks as a whole. In contrast, the 0MA fragment covers only 19 of the 259 queries in total in these two benchmarks. Our methods also apply to all the tested basic graph queries (path or tree queries) that were tested on the SNAP dataset.

In LSQB, our approach applies to 2 of the 9 queries. But as reported in Table 5.1, only 4 of the queries are equi-join queries with aggregation, the others contain joins on inequalities, which requires entirely different techniques (e.g., [92]). Of the 4 equi-join queries, 2 are not acyclic.

Our method applies to half of the equi-join aggregate queries in the TPC-DS benchmark. The queries in the benchmark are typically highly complex, often combining multiple subqueries and employing more elaborate SQL features. We observe that in some instances, $GuAO^+$ even applies to multiple subqueries in the same query. In TPC-H, our optimisations apply to 7 out of the 15 acyclic equi-join aggregate queries in the benchmark. Notably, TPC-H Q2 contains a 0MA subquery (with `MIN` aggregation) and TPC-H Q11 contains a guarded sum aggregate subquery.

TPC-H Q2 is particularly illustrative as the subquery is correlated: the attribute `p_partkey` from the outer query is used in the aggregation subquery as follows:

```
SELECT MIN(ps_supplycost)
... WHERE p_partkey = ps_partkey ...
```

The Spark SQL query planner decorrelates this subquery via typical magic decorrelation (see [141]) – resulting in the following select statement for the decorrelated subquery. This query is still guarded and thus 0MA. Our rewriting rules then apply naturally after decorrelation, with no need for any special handling of these cases.

```
SELECT ps_partkey, MIN(ps_supplycost)
... GROUP BY ps_partkey
```

We recall that our method is fully integrated into the query optimisation phase. Hence, when our optimisations are not applicable to a query, its execution is not affected. Recognizing whether the rewriting rules are applicable is trivial and requires, in our observations, negligible additional time in the query planning phase to perform our rewriting (about 2ms in all of our experiments). Going forward, it is additionally possible to pre-process queries to make them fit into the fragments where our methods are applicable.

5.3.3 Performance Impact of the Optimisations

The overall performance of our proposed optimisations on the applicable queries is summarised in Table 5.2. We refer to the reference performance of Spark SQL without any alterations as *Ref*. Our experiments on the SNAP graphs specifically are summarized in Table 5.4. The fastest execution time achieved for each case is printed in boldface. The results obtained by applications of the logical optimisations described in Section 5.1.1 for guarded queries are referred to as *Guarded Aggregate optimisation (GuAO)*. We use *GuAO*$^+$ to refer to the further extension of our logical optimisations to piecewise-guarded queries described in Section 5.1.2 plus the enhancement of the physical query plan using the AggJoin operator described in Section 5.2. The speed-up achieved by *GuAO*$^+$ over *Ref* is explicitly stated in Table 5.2 in the column *GuAO*$^+$ *Speedup*. For most benchmarks, we report end-to-end (e2e) times for subsequently executing all queries of a given benchmark where our optimisations are applicable (to the full query, or at least one subquery). In Table 5.3 we additionally report the details for all individual TPC-H queries.

Query	# joins (mean)	Ref	GuAO	GuAO$^+$	GuAO$^+$ Speedup
STATS-CEB e2e	3.33	1558±7.3	97.9±6.1	**64.8**±7.9	24.04 x
JOB e2e	7.65	3217.84±106	-	**2189.46**±76	1.47 x
TPC-H e2e SF200	1.57	3757.2	-	**3491.06**	1.08 x
TPC-H Ex.1 SF200	4	168.4	107.5	**105.11**	1.60 x
LSQB Q1 SF300	9	3096±232	**677**±23	688±23	4.57 x
LSQB Q4 SF300	3	602±37	593±15	**592**±9	1.02x
TPC-DS e2e SF100	2.52	5154.5	-	**5047.5**	1.02 x

Table 5.2: Summary of the impact of aggregate optimisation on execution times (seconds). Reported numbers are mean times over 5 runs of the same query with standard deviations given after ±. "–" indicates that the query is piecewise-guarded and, therefore, the optimisation from Section 5.1.1 for guarded queries is not applicable.

For cases where FK/PK relationships exist in the data, the columns *GuAO + FK/PK* and *GuAO*$^+$ *+ FK/PK* report the performance where this information is provided to enable the additional optimisations outlined in Section 5.1.3. The open source version of Spark SQL (Version 3.5.0) that our implementation is based on does not support specifying information about keys directly. We provide the necessary FK/PK-information via Spark SQL hints in these cases.

In all experiments, we execute each query 6 times, with the first run being a warm-up run to ensure that our measurements are not affected by initial reads of tables into memory. We report statistics gathered from the last 5 runs and report mean query execution time as well as the standard deviation over these runs. Finally, note that we execute the full query, even if our optimisation applies only to a subquery. In such a case, the plan for the subquery is optimised according to Section 5.1, and the rest of the query plan remains unchanged.

We make two key observations with respect to the performance of our methods.

When queries are challenging – e.g., they have many joins or the joins are not along PK/FK pairs – then our method provides enormous potential for speed-up. In JOB, a benchmark where suboptimal join orderings in large queries cause intermediate blow-up, we achieve almost 50% speed-up. STATS-CEB purposefully introduces joins along FK/FK relationships to challenge query evaluation systems with the resulting large number of intermediate tuples. Our method automatically avoids all of these difficulties and we see an immense 24-fold speed-up. Similarly, for the more difficult of the two LSQB queries (Q1), we see that the large number of joins creates significant intermediate blow-up with standard relational query evaluation. Again our method achieves a very large improvement of about 450%. Even in the simple query from Example 1.2.1 we observe 60% speed-up over unoptimised Spark SQL.

Query	Ref	GuAO	GuAO$^+$	GuAO$^+$ Speedup	# joins	class
TPC-H Q2 SF200	179.4±6.5	164.2±4.7	**160.6**±3.7	1.12 x	3	0MA
TPC-H Q11 SF200	361.0±13.3	346.5±9.4	**341.6**±19.2	1.06 x	2	guarded
TPC-H Ex.1 SF200	168.4±4.4	107.5±8.9	**105.11**±3.9	1.6 x	4	guarded
TPC-H Q3 SF200	798.2±179	-	**728.4**±27	1.1 x	2	pw-guarded
TPC-H Q12 SF200	**377.0**±13.1	-	381.0±11.8	0.99 x	1	pw-guarded
TPC-H Q13 SF200	440.5±11.1	-	**440.8**±4.8	1 x	1	pw-guarded
TPC-H Q16 SF200	106.1±0.95	-	**103.16**±3.6	1.03 x	1	pw-guarded
TPC-H Q17 SF200	1495.0±80.5	-	**1335.5**±65.2	1.12 x	1	pw-guarded

Table 5.3: Detailed runtimes of the TPC-H queries, including relation count and query type

	web-Google					com-DBLP				
Query	Spark	KÙZU	Neo4j	GuAO	GuAO$^+$	Spark	KÙZU	Neo4j	GuAO	GuAO$^+$
path-03	27.97±1.5	**2.14**±0.0	1686.68±114.3	6.90±0.6	6.08±0.65	6.32±1.1	**0.297**±0.0	97.80±0.7	2.35±0.5	1.59±0.12
path-04	449.14±26.9	747.15±54.4	t.o.	7.58±0.6	**6.89**±0.30	50.97±9.8	42.63±2.7	1725.10±81.6	2.24±0.4	**1.76**±0.16
path-05	o.o.m.	t.o.	t.o.	8.95±1.0	**7.53**±0.48	400.87±15.2	762.14±8.1	t.o.	2.74±0.2	**2.03**±0.25
path-06	o.o.m.	t.o.	t.o.	9.37±1.0	**8.80**±0.25	o.o.m.	t.o.	t.o.	2.98±0.2	**2.18**±0.14
path-07	o.o.m.	t.o.	t.o.	11.32±0.9	**9.76**±1.21	o.o.m.	t.o.	t.o.	3.64±0.2	**2.38**±0.26
path-08	o.o.m.	t.o.	t.o.	11.30±2.1	**10.05**±1.49	o.o.m.	t.o.	t.o.	3.75±0.4	**2.53**±0.30
tree-01	539.11±22.4	t.o.	t.o.	7.73±1.0	**6.53**±1.11	25.96±4.5	99.16±0.9	t.o.	1.95±0.1	**1.47**±0.28
tree-02	o.o.m.	t.o.	t.o.	12.43±3.2	**7.29**±0.73	328.88±11.5	t.o.	t.o.	3.02±0.7	**1.69**±0.16
tree-03	o.o.m.	t.o.	t.o.	12.21±5.6	**8.16**±0.66	o.o.m.	t.o,	t.o.	3.17±0.2	**1.99**±0.16

Table 5.4: Performance on SNAP graphs, compared to graph database systems

Query	Ref	GuAO	GuAO$^+$	GuAO$^+$ Speedup	GuAO + FK/PK	GuAO$^+$ + FK/PK
STATS-CEB e2e	1558±7.3	97.9±6.1	**64.8**±7.9	24.04 x	**58.9**±0.6	64.0±7.7
TPC-H Q11 SF200	361.0±13.3	346.5±9.4	**341.6**±19.2	1.06 x	350.1±11.6	344.7±13.9
TPC-H V.1 SF200	168.4±4.4	107.5±8.9	**105.11**±3.9	1.6 x	106.1±1.1	**102.04**±3.9
LSQB Q1 SF300	3096±232	**677**±23	688±23	4.57 x	688±41	689±35
LSQB Q4 SF300	602±37	593±15	**592**±9	1.02x	**587**±11	600±21

Table 5.5: Comparison of GuAO, GuAO$^+$, with and without the FK/PK optimisations, where applicable

On the other hand, we observe that especially the two TPC benchmarks contain primarily queries where the join evaluation itself is very straightforward. In TPC-H, 4 of the 7 tested

queries contain only a single join (see Table 5.3). In TPC-DS we observe similar patterns. As a result, there is little to no unnecessary materialisation in many of these queries. Our key insight here is that the experiments confirm that our method (and in particular *GuAO*$^+$) does not introduce any overhead in these cases. By not causing performance decrease in those cases where there is no unnecessary materialisation, combined with some gains in the few harder queries, we still see modest overall speed-ups for these benchmarks.

With respect to basic graph queries, we see in Table 5.4 that even with significant resources, counting short paths and small trees is effectively impossible on large graphs with current methods. This holds for both, Spark SQL and for specialised graph database systems. In stark contrast, *GuAO* and *GuAO*$^+$ effectively trivialize these types of queries even with significantly less resources than are available on our test system (the highest observed memory usage for *GuAO*$^+$ in our SNAP experiments was roughly 5GB). Since these experiments focus on graph data, we additionally compared with specialised graph database systems (Neo4j [120], KÙZU [89], and GraphDB [69]). The results of these experiments are equally sobering as with standard Spark SQL, in that almost all queries failed with timeout (set to 30 min). This is in sharp contrast to GuAO and GuAO$^+$, which answer these queries in a matter of a few seconds.

To evaluate the effect of FK/PK information we extended a subset of our tests from Section 5.3, by manually providing FK/PK information, and applying the further optimisations from Section 5.1.3. The results for these experiments are summarised in Table 5.5. We make two important observations from our experimental results for *GuAO + FK/PK* and *GuAO*$^+$ *+ FK/PK*. First, in the case of *GuAO*$^+$, there is little to no gain from the simplification to semi-joins. This confirms that in cases where joins follow FK/PK relationships, the physical operators from Section 5.2 are in practical terms as efficient as semi-joins. In a sense, this means that using *GuAO*$^+$ makes it unnecessary to be aware of FK/PK relationships on join attributes, as the execution is implicitly optimised appropriately in this scenario anyway. Detailed analysis of the data shows that the small performance differences, in both directions, are primarily due to the potential additional initial grouping operations as described in Section 5.1.3. The only case where we observe noteworthy improvement is for *GuAO+FK/PK* on STATS-CEB, where the additional use of FK/PK information yields a 60% speed-up over *GuAO*. The potential for improvements through FK/PK information seems highly data- and query-dependent. Overall, we conclude that, in the context of our method, FK/PK information is less relevant than might be expected.

In summary, our experiments paint a clear picture. In more challenging queries, our approach offers very significant improvements. At the same time, in cases where little unnecessary materialisation is performed, *GuAO*$^+$ introduces no additional overhead and thus exhibits no performance degradation on simpler queries.

5.3.4 Quantifying the Reduction of Materialisation

Throughout this chapter, we have been motivated by the premise that easy to implement logical optimisation rules for query plans can avoid a significant amount of intermediate materialisation in aggregate queries. Moreover, with the addition of natural physical operators, we can avoid any such materialisation altogether. However, this raises the question of how much unnecessary materialisation actually occurs when using standard query planning methods.

To study this question, we compare the maximum number of tuples that occur in an intermediate table during query execution for the STATS-CEB queries. Again, we report the mean over 5 runs (we omit error bars as the variation between runs is mostly 0 and negligible in other cases). We note that these intermediate table sizes are naturally closely correlated to overall memory consumption, as well as communication cost in a distributed setting. Figure 5.6 reports the peak number of materialised tuples during query execution for the 20 queries where standard Spark SQL materialises the most intermediate tuples. The data clearly shows that an improvement in the order of magnitudes of materialised tuples is often possible. In particular, we see the well documented effect of classical relational query processing techniques, leading to substantial intermediate blow-up. The largest relations in the dataset have in the order of $3 \cdot 10^5$ tuples, an enormous difference to the observed sizes of up to 10^{10} intermediate tuples for *Ref*. The data shows that by rewriting the logical query plans according to Section 5.1, we regularly see a reduction in peak intermediate table size of over 2 orders of magnitude.

However, the optimised logical query plan still requires some mild materialisation between aggregation steps, which we manage to eliminate with the physical operators described in Section 5.2. The resulting *GuAO*$^+$ system consistently reduces the number of materialised tuples by **at least 3 orders of magnitude** on the reported queries in Figure 5.6. In fact, the reported numbers for *GuAO*$^+$ are always precisely the cardinality of the largest relation in the query, as the execution using our method never introduces any new tuples (cf. Section 5.2). That is, this number can also inherently not be improved upon. Over the whole benchmark, we observe that the peak number of materialised tuples by *GuAO*$^+$ is *at least 10 times less* than that of standard Spark SQL query execution in 118 out of the 146 queries. In all other cases, the peak number of materialised tuples by *Ref* and *GuAO*$^+$ is exactly the same, i.e., *Ref* is never better.

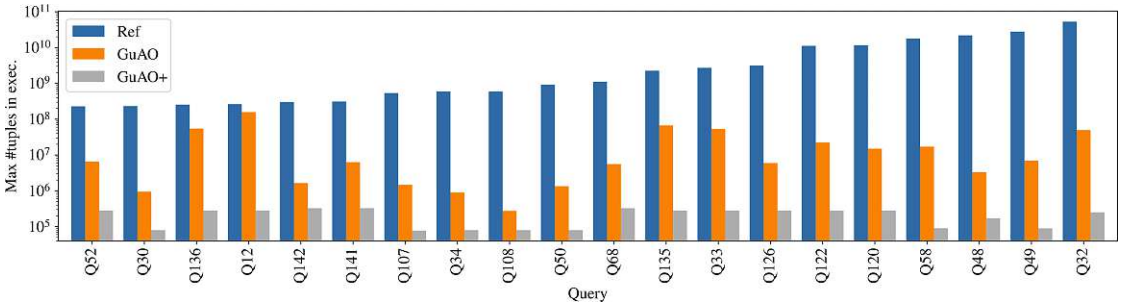


Figure 5.6: Comparison of the maximal number of materialised tuples in a table during query execution for 20 queries of STATS-CEB. Y-axis in logarithmic scale (base 10).

5.4 Extension to Unguarded Queries

Our next goal is to investigate, first of all, whether it is feasible at all to extend our optimisations beyond piecewise-guardedness with performance benefits. Additionally, we would like to consider various "degrees" of unguardedness to determine how far we can go efficiently.

We start by considering an example of an unguarded query based on the TPC-H schema, similar to the query in Figure 1.2.

```
SELECT COUNT(*), p_brand, r_name
FROM part, partsupp, supplier,
  nation, region
WHERE p_partkey   = ps_partkey
  AND s_suppkey   = ps_suppkey
  AND n_nationkey = s_nationkey
  AND r_regionkey = n_regionkey
  AND p_price >
   (SELECT avg (p_price) FROM part)
AND r_name IN ('Europe', 'Asia')
GROUP BY p_brand, r_name
```

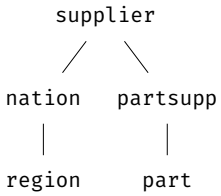


Figure 5.7: Unguarded query over the TPC-H schema and its corresponding join tree

In this example, we compute the count of rows grouped by the `p_brand` and `r_name` attributes of two different relations `part` and `region`. Clearly, there is no way around performing at least one join and materialising the output, as the output per definition contains the distinct projection to `p_brand` and `r_name` over the joins of the query. However, we can still do significantly better than performing only joins, materialising the full enumeration, and performing the grouping and aggregation operation afterwards.

5.4.1 Introducing the GroupAggJoin physical operator

To extend our approach for answering piecewise-guarded aggregate queries, we require a way to propagate not only frequencies and aggregates up the join tree, but also grouping attributes. To this end, we introduce a new physical operator: *GroupAggJoin*. The GroupAggJoin operator not only performs aggregation inside the join loop but also grouping via specified grouping attributes, outputting the grouping attributes as well as aggregate results.

Algorithm 5.3: Hash Join with grouping and aggregate propagation

Input: A streamed tuple r from the list R and a list S of tuples with the same values of the join attributes;
List $I = \{s_1, \ldots, s_n\}$ of indices of aggregate attributes Agg_{s_i};
List $G = \{s_1, \ldots, s_n\}$ of indices of grouping attributes G_{s_i};

```
1  Function GroupAggHashJoin(r, S, I, G)
2      groups ← Map;
3      foreach s ∈ S do
4          if groups[s[G]] exists then
5              g ← groups[s[G]];
6          else
7              g ← {c : 0};
8              foreach i ∈ I do
9                  if A_i ∈ {MIN, MAX} then g[i] ← init[s];
10                 if A_i ∈ {SUM, COUNT} then g[i] ← 0;
11             end
12         end
13         g.c ← g.c + s.c;
14         foreach i ∈ I do
15             if A_i ∈ {MIN, MAX} then s[Agg_i] ← A(g[Agg_i], s.Agg_i);
16             if A_i ∈ {SUM, COUNT} then s[Agg_i] ← g[Agg_i] + s.Agg_i;
17         end
18     end
19     foreach (k, v) ∈ groups do
20         v.c ← v.c · r.c
21         foreach i ∈ I do
22             if A_i ∈ {SUM, COUNT} then v[Agg_i] ← v[Agg_i] · r.c;
23         end
24         emit (r, k, v);
25     end
```

In Algorithm 5.3 we provide simplified pseudocode describing the hash join variant of the GroupAggJoin. For compactness, we describe the version of the algorithm where we iterate over individual tuples from the R relation and read the matching tuples from S in

chunks. The key modification required for the extension to grouping is the introduction of a map (*groups*), mapping the grouping keys to an aggregation buffer. After initialising the grouping map, we iterate over the tuples $s \in S$. By $s[G]$ we denote the projection of the tuple s to the grouping attributes G. First, we check if a grouping buffer exists for $s[G]$, and if so, retrieve it, else initialise it similarly to the GroupAggJoin operator described in Algorithm 5.1. Next, since we still need to keep track of frequencies, we add the tuple frequency $s.c$ to the grouping buffer's frequency value $g.c$. Then, for each aggregate attribute $i \in I$, we proceed similar to Algorithm 5.1, updating the group's aggregation buffer. Finally, we can iterate over the aggregation buffers. After multiplying the frequency values by $r.c$, we emit the tuple r, the grouping attributes themselves (k), and the final aggregation buffer (v).

As in the case of the AggJoin, we implement 3 types of GroupAggJoin based on Spark SQL's broadcast-hash-join, shuffled-hash-join, and (sort-)merge-join. Details for the merge-join are omitted here. An extension of the pseudocode presented in Section 5.2 is however straightforward.

5.4.2 Benchmarks for Unguarded Queries

To the extent of our knowledge, no benchmarks exist for specifically evaluating the challenging class of unguarded queries. Therefore, we decide to introduce a new benchmark based on the Join-Order-Benchmark(JOB) [101], which we refer to as *JOBUnguarded*. We choose 33 diverse (in the number of joins) JOB queries, and, based on these construct up to 8 new unguarded queries. The queries from the JOB benchmark do not contain GROUP BY clauses, and are piecewise-guarded. Therefore, in order to obtain unguarded queries, we extend the queries by GROUP BY clauses (and the corresponding output attributes). We refer to a query with k unguarded attributes as k-unguarded. We apply the following for constructing a k-unguarded query:

- Check whether the query contains at least k relations. In this case, proceed.
- Choose k attributes, $a_1, ..., a_k$ which are
  - **never** part of the same relation as another attribute
  - not part of the filter conditions (preferably)
  - not join keys (preferably)
- Add a clause `GROUP BY` $a_1, ..., a_k$ and add $a_1, ..., a_k$ to the SELECT clause.

For each JOB query, we start by constructing a 2-unguarded query, up to a 9-unguarded query. In total, we obtain 193 unguarded queries. In Table 5.6, we provide an overview of the number of k-unguarded queries generated.

For example, the following 3-unguarded query, `3a-unguarded-3`, was derived from the JOB query 3a with 4 relations by adding the 3 unguarded attributes `mi.id, mk.id` and `k.phonetic_code`, from 3 different relations.

```
SELECT MIN(t.title) AS movie_title,
       mi.id, mk.id, k.phonetic_code
FROM keyword AS k,
     movie_info AS mi,
     movie_keyword AS mk,
     title AS t
WHERE k.keyword LIKE '%sequel%'
  AND mi.info IN ('Sweden',
                  'Norway',
                  'Germany',
                  'Denmark',
                  'Swedish',
                  'Denish',
                  'Norwegian',
                  'German')
  AND t.production_year > 2005
  AND t.id = mi.movie_id
  AND t.id = mk.movie_id
  AND mk.movie_id = mi.movie_id
  AND k.id = mk.keyword_id
GROUP BY mi.id, mk.id, k.phonetic_code;
```

5.4.3 Performance Evaluation

We benchmark the extended implementation including the GroupAggJoin operator on the JOBUnguarded benchmark. The runtime measurements are performed on a VM with a 32-core AMD EPYC-Milan CPU with 256 GB RAM (we require less RAM than for the previous experiments since there is no risk of running out of memory for these queries). The benchmark environment and queries are provided at `https://github.com/arselzer/spark-eval-groupagg`. Again, we perform, both for the original Spark SQL implementation and the optimised version, a "warm-up" run followed by 5 runs, of which we take the average.

A summary of the end-to-end runtimes including e2e speedup per group of k-unguarded queries is given in Table 5.6. Furthermore, we show how many of the individual queries were sped up by the optimisation (#Speedups%). It is to be noted that the total e2e runtimes decrease since the number of queries (#queries) decreases with increasing number of unguarded attributes (as there are fewer queries from which we can construct unguarded variants).

Ung. Atts	#queries	Ref	Opt	Speedup	#Speedups%
2	32	755.71	**597.78**	1.26x	69%
3	32	759.60	**586.31**	1.30x	78%
4	31	746.84	**593.58**	1.26x	68%
5	27	690.61	**550.40**	1.25x	78%
6	22	537.93	**484.21**	1.11x	78%
7	22	**559.50**	583.72	0.96x	68%
8	16	351.10	**325.41**	1.08x	69%
9	11	248.17	**232.06**	1.07x	73%

Table 5.6: Comparison of the Spark SQL reference implementation vs. the optimised version, for JOBUnguarded.

We observe clear speedups in e2e-runtime of 1.25x-1.30x between 2- and 5-unguarded queries. The number of speedups achieved on individual queries is in all cases at least 68%. From 6-unguarded queries on, we can observe a significant degradation in performance, even resulting in a slight slowdown for 7-unguarded queries. Overall, however, even on these queries the performance ranges from comparable to slightly better.

In Figure 5.8, we present the runtimes of the 2-unguarded queries in detail. We also provide individual comparisons for 3- to 9-unguarded queries in Figures 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, and 5.15.

As visible in Figure 5.8, the optimised implementation provides the greatest benefit for hard instances. In fact, for the 15 hardest instances (sorted by reference runtime), our implementation always outperforms plain Spark SQL.

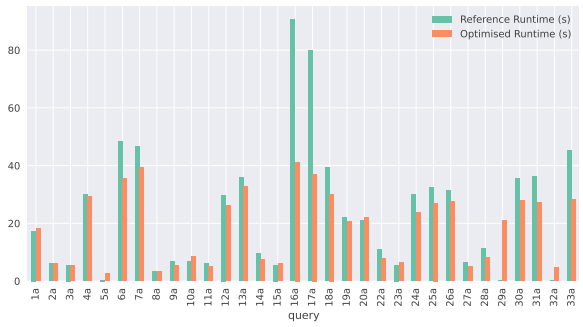


Figure 5.8: Queries with 2 unguarded attributes

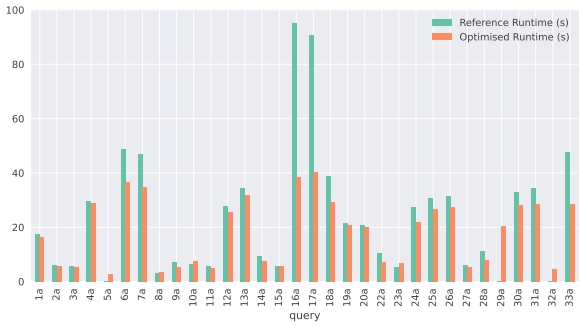


Figure 5.9: Queries with 3 unguarded attributes

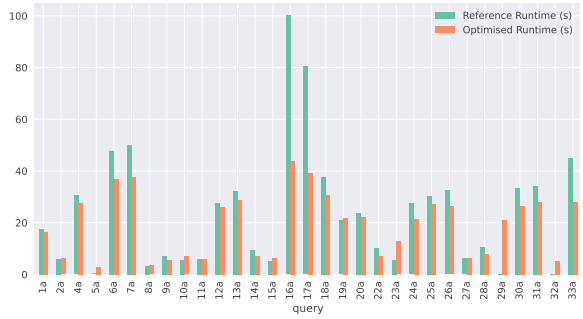


Figure 5.10: Queries with 4 unguarded attributes

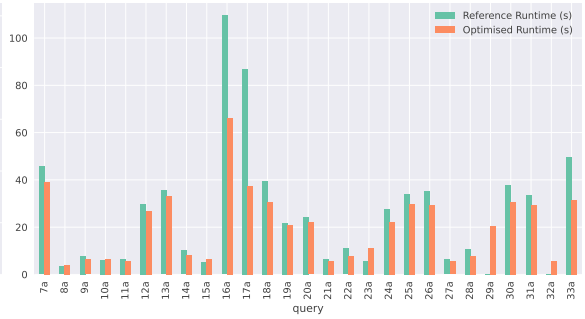


Figure 5.11: Queries with 5 unguarded attributes

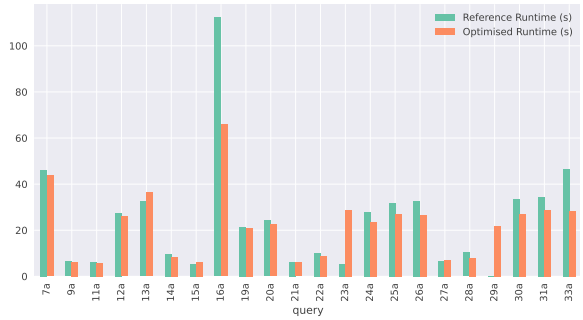


Figure 5.12: Queries with 6 unguarded attributes

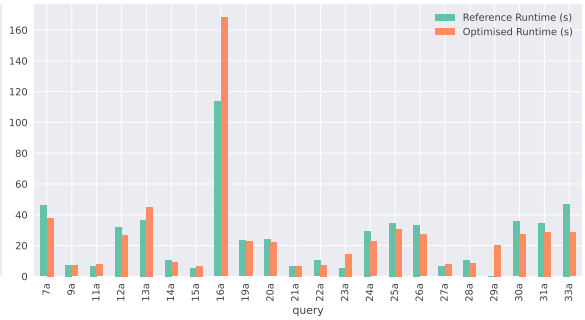


Figure 5.13: Queries with 7 unguarded attributes

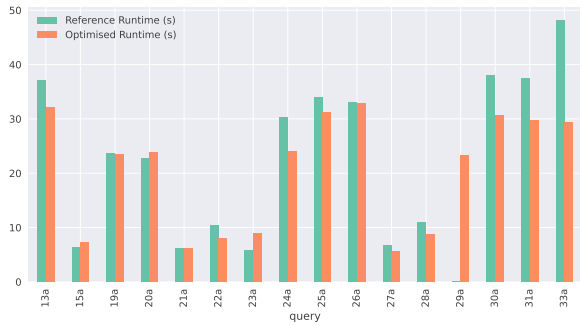


Figure 5.14: Queries with 8 unguarded attributes

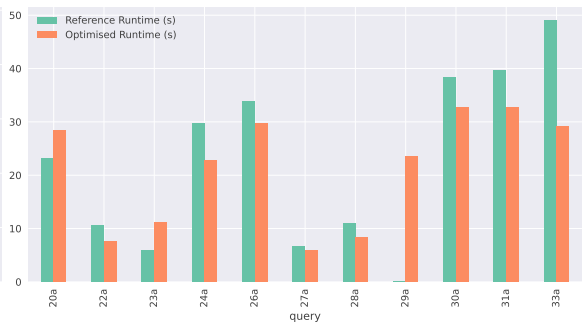


Figure 5.15: Queries with 9 unguarded attributes

5.5 Summary

In this chapter, we have identified the broad classes of guarded and piecewise-guarded queries. We have introduced several optimisations for guarded aggregate queries, enabling

significant reductions of the need to materialise intermediate results when evaluating analytical queries with aggregates over join or path queries. We have integrated our optimisations into Spark SQL, which has been specifically designed as a powerful tool to deal with complex analytical queries. Our experimental evaluation confirms that the proposed techniques can provide significant performance improvements by avoiding costly materialisation, especially in larger queries. Additionally, we propose the use of new physical operators that extend semi-joins to manage frequencies and other aggregate information with the aim to completely eliminate the computation of intermediate joins and to facilitate straightforward integration into physical query plans. Finally, we have extended the physical operator and the logical optimisation to handle unguarded queries. By introducing a new benchmark for unguarded queries, we show that the approach can handle moderately-unguarded queries effectively.

CHAPTER 6

Cyclic Join Queries – From Theory to Practice

With structure-guided query processing of most acyclic aggregate queries covered, we focus on cyclic queries in this chapter. We will make steps towards bringing decomposition-based techniques into practice by introducing a flexible approach towards the computation of decompositions and implementing a query-processing pipeline based on it.

We revisit the notion of candidate tree decompositions from [60] in Section 6.1. In Section 6.2, we introduce soft hypertree decompositions and soft hypertree width (*shw*). The integration of preferences and constraints into soft hypertree decompositions is discussed in Section 6.3 and the implementation of the system, including the development of cost functions, is reported on in Section 6.4. Experimental results with these decompositions are reported in Section 6.5. We summarise the results of this Chapter in Section 6.6.

6.1 Revisiting Candidate Tree Decompositions

In this section, we revisit the problem of constructing tree decompositions by selecting the bags from a given set of candidate bags. This leads us to the notion of candidate tree decompositions (CTDs) and the CandidateTD problem, which we both define next.

Definition 6.1.1. *Let H be a hypergraph and let $\mathbf{S} \subseteq 2^{V(H)}$ be a set of vertex sets of H (the so-called* "candidate bags"*). A* candidate tree decomposition (CTD) *of $\mathbf{S}$ is a tree decomposition (T, B) of H, such that, for every node u of T, $B(u) \in \mathbf{S}$ [60].*

In the CandidateTD problem, we are given a hypergraph H and a set $\mathbf{S} \subseteq 2^{V(H)}$ (= the set of candidate bags), and we have to decide whether there exists a tree decomposition (T, B) of H, such that, for every node $u \in T$, we have $B(u) \in \mathbf{S}$.

The idea of generating TDs from sets of candidate bags was heavily used in a series of papers by Bouchitté and Todinca [27, 28, 29, 30, 31] in a quest to identify a large class of graphs for which the treewidth (and also the so-called minimum fill-in, for details see any of the cited papers) can be computed in polynomial time. More specifically, this class of graphs G is characterised by having a polynomial number of minimal separators (i.e., subsets S of $V(G)$, such that two vertices u, v of G are in different connected components of the induced subgraph $G[V(G) \setminus S]$ but they would be in the same connected component of $G[V(G) \setminus S']$ for every proper subset $S' \subset S$). The key to this tractability result was to precisely identify and compute in polynomial time the set of candidate bags via the minimal separators and the so-called potential maximal cliques of a given graph. Ravid et al. [135] applied these ideas to efficiently enumerate TDs under a monotonic cost measure, such as treewidth.

It was shown in [60], that the problem of deciding whether a hypergraph has width $\leq k$ for various notions of decompositions (in particular, the generalised hypertree width *ghw* and the so-called fractional hypertree width *fhw*) can be reduced to the CandidateTD problem by an appropriate choice of the set S of candidate bags. However, it was also shown in [60] that some care is required as far as the form of the sought after TDs is concerned. In particular, it was shown in [60], that, in the unrestricted form given in Definition 6.1.1, the CandidateTD problem is NP-complete. In order to ensure tractability, the following restriction on TDs was formally defined in [60]:

Definition 6.1.2. *A rooted tree decomposition* (T, B) *is in* component normal form (CompNF)*, if for each node* $u \in V(T)$*, and for each child* c *of* u*, there is exactly one* $[B(u)]$*-component* C_c *such that* $B(T_c) = \bigcup C_c \cup (B(u) \cap B(c))$ *[60].*

Alternatively, we could define CompNF by making use of the fact that the TDs of a hypergraph H are exactly the TDs of its Gaifmann graph $G(H)$ (see Chapter 2). The CompNF condition given in Definition 6.1.2 can then be rephrased as requiring that, for a node u and child node c in a rooted TD (T, B), that the vertices in $B(T_c)$ must not be separated by the vertex set $B(u) \cap B(c)$.

It was shown in [60] that the CandidateTD problem becomes tractable if we ask for the existence of a CTD *in CompNF*. We note that also the algorithms presented in the works of Bouchitté and Todinca (see, in particular, [29, 30]) as well as in [135] implicitly only consider TDs in CompNF. From now on, we consider the CandidateTD problem only in this restricted form. By slight abuse of notation, we will simply refer to it as the CandidateTD problem, with the understanding that we are only looking for CTDs in CompNF. Gottlob et al. [60] presented a poly-time algorithm, which we recall in a slightly adapted form in Algorithm 6.1, for deciding the CandidateTD problem.

To discuss Algorithm 6.1 in detail, we first have to define the terminology used in the algorithm, namely the notions of a "block", a "basis", and what it means that a block is "satisfied". We call a pair (S, C) of *disjoint* subsets of $V(H)$ a *block* if C is a maximal set of $[S]$-connected vertices of H or $C = \emptyset$. We say that (S, C) is *headed* by S. For

Algorithm 6.1: CompNF Candidate Tree Decomposition of $\mathbf{S}$

input: Hypergraph H and a set $\mathbf{S} \subseteq 2^{V(H)}$.
output: "Accept", if there is a CompNF CTD for $\mathbf{S}$
"Reject", otherwise.

```
1  blocks = all blocks headed by any S ∈ S ∪ {∅}
2  foreach (S, C) ∈ blocks do
3  |  if C = ∅ then basis(S, C) ← ∅
4  |  else basis(S, C) ← ⊥   /* a block is satisfied iff its basis is not ⊥
   |   */
5  end
6  repeat
7  |  foreach (S, C) ∈ blocks that are not satisfied do
8  |  |  foreach X ∈ S \ {S} do
9  |  |  |  if X is a basis of (S, C) then
10 |  |  |  |  basis(S, C) ← X
11 |  |  |  end
12 |  |  end
13 |  end
14 until no blocks changed
15 if basis(∅, V(H)) ≠ ⊥ then
16 |  return Accept
17 end
18 return Reject
```

two blocks (S, C) and (X, Y) define $(X, Y) \leq (S, C)$ if $X \cup Y \subseteq S \cup C$ and $Y \subseteq C$. Note that the notion of a *block* was already introduced in the works of Bouchitté and Todinca [27, 28, 29, 30, 31] and later used by Ravid et al. [135]. In this work (following [60]), blocks are defined slightly more generally in the sense that Bouchitté and Todinca only considered blocks (S, C) where S is a minimal separator of the given graph (rather than an arbitrary, possibly even empty, subset of the vertices) and C is not allowed to be empty.

A block (S, C) is *satisfied* if a CompNF TD of $H[S \cup C]$ exists with root bag S. If $C = \emptyset$, satisfaction is trivial. Finally, for a block (S, C) and $X \subseteq V(H)$ with $X \neq S$, let $(X, Y_1), \ldots, (X, Y_\ell)$ be all the blocks headed by X with $(X, Y_i) \leq (S, C)$. Then we say that X *is a basis of* (S, C) if the following conditions hold:

1. $C \subseteq X \cup \bigcup_{i=1}^{\ell} Y_i$.
2. For each $e \in E(H)$ such that $e \cap C \neq \emptyset$, $e \subseteq X \cup \bigcup_{i=1}^{\ell} Y_i$.
3. For each $i \in [\ell]$, the block (X, Y_i) is satisfied.

The concepts of a *block* and of a *basis of a block* are related to a CTD (in CompNF) in the following way: Recall from Definition 6.1.2 that if u, c are parent-child nodes in a TD, then there is exactly one $[B(u)]$-component C_c such that $B(T_c) = \bigcup C_c \cup (B(u) \cap B(c))$. Now consider the subtree T' of T that consists of the node u plus the entire subtree T_c. Moreover, consider the block (S, C) with $S = B[u]$ and $C = \bigcup C_c \setminus B(u)$. Then this block is indeed *satisfied* by taking as CTD of $H[S \cup C]$ the subtree T' of T (where the bag of each node in T' is exactly the bag of the corresponding node in T). The goal of Algorithm 6.1 is to satisfy progressively larger blocks, increasing $|S \cup C|$. If $(\emptyset, V(H))$ is satisfied, a CTD for H exists, as checked in Line 11.

The relationship between the notions of *block*, *basis*, and *satisfaction* of a block is summarised by the following property that was proved in [60]: *Let (S, C) be a block and let X be a basis of (S, C), then (S, C) is satisfied.* The proof idea of this property is as follows: let $(X, Y_1), \ldots, (X, Y_\ell)$ be all the blocks headed by X with $(X, Y_i) \leq (S, C)$. By the third condition of a basis, the block (X, Y_i) is satisfied for each $i \in [\ell]$. Hence, there exists a CTD (T_i, B_i) of each subhypergraph $H[X \cup Y_i]$, such that the bag of the root is X for all theses CTDs. We can merge all these root nodes into a single node to form a single TD (T', B') with the T_i's as subtrees. A CTD for $H[S \cup C]$ is then obtained by taking as root a node with bag S and appending the root of T' (with bag X) plus the entire tree T'. The aim of the repeat-loop in Algorithm 6.1 (Lines 5-10) is precisely to check if we can mark yet another block (S, C) as satisfied by identifying a basis for it. Initially, in the foreach-loop on Lines 2-4, only the blocks with empty C-part are assigned the trivial basis $\emptyset$ and thus marked as satisfied. Finally, on Lines 11-13, the algorithm returns "Accept" (i.e., a CTD of H exists) if the block $(\emptyset, V(H))$ is marked as satisfied via a non-empty basis. Otherwise, it returns "Reject". The bottom-up approach in Algorithm 6.1, constructing a CTD by combining subtrees, also appears similarly in prior work [30, 135].

For the polynomial-time complexity of Algorithm 6.1, the crucial observation is that the number b of blocks (S, C) is bounded by $|\mathbf{S}| * |V(H)|$, i.e., for each $S \in \mathbf{S}$, there cannot be more components C than vertices in H. As a coarse-grained upper bound on the complexity of Algorithm 6.1, we thus get $b^4 * ||H||$, i.e., each of the 3 levels of nested loops has at most b iterations and the cost of checking the basis-property on Line 8 can be bounded by b times the size of (some representation of) H.

6.2 Soft Hypertree Width

So far, it is not known whether there is a set of candidate bags $\mathbf{S}_{H,k}$ for a hypergraph H such that there is a CTD for $\mathbf{S}_{H,k}$ if and only if $hw(H) \leq k$. However, in [58], it was shown that there always exists a HD of minimal width such that all bags of nodes c with parents p are of the form $B_c = (\bigcup \lambda_c) \cap (\bigcup C_p)$ where C_p is a $[\lambda_p]$-component of H. In principle, this gives us a concrete way to enumerate a sufficient list of candidate bags. The number of all such bags is clearly polynomial in H: there are at most $|E(H)|^{k+1}$ sets of at most k edges, and each of them cannot split H into more than $|E(H)|$ components.

The only point that is unclear when enumerating such a list of candidate bags, is how to decide beforehand whether two sets of edges λ_c, λ_p are in a parent/child relationship (and if so, which role they take). However, we observe that the parent/child roles are irrelevant for the polynomial bound on the number of candidate bags. Hence, we may drop this restriction and instead simply consider all such combinations induced by any two sets of at most k edges. Concretely, this leads us to the following definitions.

Definition 6.2.1 (The set $\mathsf{Soft}_{H,k}$)**.** *For hypergraph H, we define $\mathsf{Soft}_{H,k}$ as the set that contains all sets of the form*

$$B = \left(\bigcup \lambda_1\right) \cap \left(\bigcup C\right) \tag{6.1}$$

where C is a $[\lambda_2]$-component of H and λ_1, λ_2 are sets of at most k edges of H.

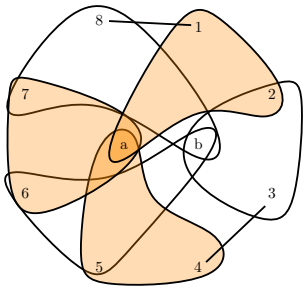
Definition 6.2.2 (Soft Hypertree Width)**.** *A* soft hypertree decomposition *of width k for hypergraph H is a candidate tree decomposition for $\mathsf{Soft}_{H,k}$. The* soft hypertree width *(shw) of hypergraph H is the minimal k for which there exists a soft hypertree decomposition of H.*

This measure naturally generalises the notion of hypertree width in a way that remains tractable to check (for fixed k), but removes the need for the special condition.

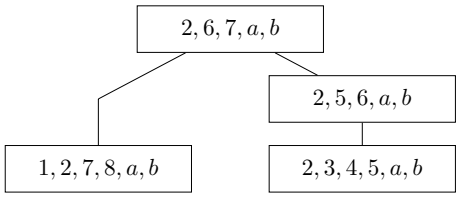
Theorem 6.2.1. *Let $k \geq 1$. Deciding, for a given hypergraph H, whether $shw(H) \leq k$ holds, is feasible in polynomial time in the size of H. The problem even lies in the highly parallelisable class* LogCFL*.*

Proof Sketch. In [62], it is shown that checking if $hw(H) \leq k$ holds for fixed $k \geq 1$ is in LogCFL. The key part of the proof is the construction of an alternating Turing machine (ATM) that runs in Logspace and Ptime. The ATM constructs an HD in a top-down fashion. In the existential steps, one guesses a λ-label of the next node in the HD. Given the label λ_c of the current node and the label λ_p of its parent node, one can compute the set of $[\lambda_c]$-components that lie inside a $[\lambda_p]$-component. In the following universal step, the ATM has to check recursively if all these $[\lambda_c]$-components admit an HD of width $\leq k$.

This ATM can be easily adapted to an ATM for checking if $shw(H) \leq k$ holds. The main difference is, that rather than guessing the label λ_c of the current node (i.e., a collection of at most k edges), we now simply guess an element from $\mathsf{Soft}_{H,k}$ as the bag B_c of the current node. The universal step is then again a recursive check for all components of B_c that lie inside a $[B_p]$-component, if they admit a candidate TD of width $\leq k$. The crucial observation is that we only need Logspace to represent the bags $B \in \mathsf{Soft}_{H,k}$, because every such bag is uniquely determined by the label λ_c and a $[\lambda_p]$-component. Clearly, λ_c can be represented by up to k (pointers to) edges in $E(H)$, and a $[\lambda_p]$-component can be represented by up to $k+1$ (pointers to) edges in $E(H)$, i.e., up to k edges in λ_p plus 1 edge from the $[\lambda_p]$-component. Clearly, the latter uniquely identifies a component, since no edge can be contained in 2 components. □



(a) Hypergraph H_2 with $ghw(H_2) = shw(H_2) = 2$ and $hw(H_2) = 3$. Some edges are coloured for visual clarity.



(b) A soft hypertree decomposition of H_2 with width 2.

Figure 6.1: (a) A hypergraph H_2 and (b) its soft hypertree decomposition.

The main argument in the proof of Theorem 6.2.1 was that the ATM for checking $hw(H) \leq k$ can be easily adapted to an algorithm for checking $shw(H) \leq k$. Actually, the straightforward adaptation of existing *hw*-algorithms to *shw*-algorithms is by no means restricted to the rather theoretical ATM of [62]. The parallel algorithm log-k-decomp from [58] performs significant additional effort to control orientation of subtrees in order to guarantee the special condition. This is unnecessary for *shw*-computation, where the orientation of subtrees is irrelevant. Instead, we may follow the philosophy of the much simpler BalancedGo algorithm in [65] for *ghw*-computation. By a standard argument (see e.g., [58, Lemma 3.14]), a CTD for $\mathsf{Soft}_{H,k}$ always contains a *balanced separator*. Hence, one can simply adapt BalancedGo algorithm to only consider separators in $\mathsf{Soft}_{H,k}$ to obtain another algorithm for checking *shw* that is suitable for parallelisation. In Section 6.3, we will extend the CTD-framework by constraints and preferences. It will turn out that we can thus also capture the opt-k-decomp approach from [136], that integrates a cost function into the computation of HDs. To conclude, the key techniques for modern decomposition algorithms are also applicable to *shw*-computation.

The relationship of $shw(H)$ with $hw(H)$ and *ghw* is characterised by the following result:

Theorem 6.2.2. *For every hypergraph H, the relationship $ghw(H) \leq shw(H) \leq hw(H)$ holds. Moreover, there exist hypergraphs H with $shw(H) < hw(H)$.*

Proof. By [58], if $hw(H) = k$, then there exists a hypertree decomposition of width k such that every bag is of the form from Equation (6.1). That is, for every HD (T, λ, χ), we immediately get a candidate TD (T, χ) for $\mathsf{Soft}_{H,k}$. Hence, $shw(H) \leq hw(H)$. Furthermore, every bag in $\mathsf{Soft}_{H,k}$ is subset of a union of k edges, hence $\rho(B) \leq k$ for any B from Equation (6.1). Thus, also $ghw(H) \leq shw(H)$. In the example below, we will present a hypergraph H with $shw(H) < hw(H)$. □

Example 6.2.3. *Let us revisit the hypergraph H_2 from [6], which was presented there to show that ghw can be strictly smaller than hw. The hypergraph is shown in Figure 6.1a.*

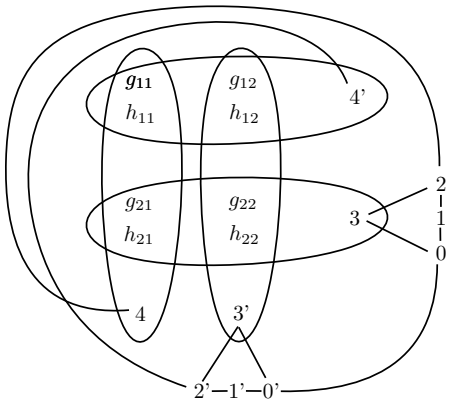


Figure 6.2: The hypergraph H_3 (some edges omitted) with $ghw(H_3) = shw(H_3) = 3$ and $hw(H_3) = 4$ (adapted from [5]).

It consists of the edges $\{1,8\},\{3,4\},\{1,2,a\},\{4,5,a\},\{6,7,a\},\{2,3,b\},\{5,6,b\},\{7,8,b\}$ *and no isolated vertices. It is shown in [6] that* $ghw(H_2) = 2$ *and* $hw(H_2) = 3$*. We now show that also* $shw(H_2) = 2$ *holds.*

A candidate tree decomposition for $\mathsf{Soft}_{H_2,2}$ *is shown in Figure 6.1b. Let us check that* $\mathsf{Soft}_{H_2,2}$ *contains all the bags in the decomposition. The bags* $\{1,2,7,8,a,b\}$ *and* $\{2,3,4,5,a,b\}$ *are the union of 2 edges and thus clearly in the set. The bag* $\{2,6,7,a,b\}$ *is induced by* $\lambda_2 = \{\{3,4\},\{2,3,b\}\}$*. There is only one* $[\lambda_2]$*-component* C*, which contains all edges of* $E(H) \setminus \lambda_2$*. Hence,* $\bigcup C$ *contains all vertices in* $V(H) \setminus \{3\}$*. We thus get the bag* $\{2,6,7,a,b\}$ *as* $(\bigcup \lambda_1) \cap (\bigcup C)$ *with* $\lambda_1 = \{\{2,3,b\},\{6,7,a\}\}$*. The remaining bag* $\{2,5,6,a,b\}$ *is obtained similarly from* $\lambda_2 = \{\{1,8\},\{1,2,a\}\}$ *which yields a single component* C *with* $\bigcup C = H(H) \setminus \{1\}$*. We thus get the bag* $\{2,5,6,a,b\}$ *as* $(\bigcup \lambda_1) \cap (\bigcup C)$ *with* $\lambda_1 = \{\{1,2,a\},\{5,6,b\}\}$*.*

We have chosen the hypergraph H_2 above, since it has been the standard (and only) example in the literature of a hypergraph with $ghw = 2$ and $hw = 3$. It illustrates that the relaxation to *shw* introduces useful new candidate bags.

We now present an example of a hypergraph with $shw = 3$ and $hw = 4$. To this end, We consider the hypergraph H_3 adapted from [5], defined as follows. Let $G = \{g_{11}, g_{12}, g_{21}, g_{22}\}$, $H = \{h_{11}, h_{12}, h_{21}, h_{22}\}$, and $V = \{0, 1, 2, 3, 4, 0', 1', 2', 3', 4'\}$. The vertices of H_3 are the set $H \cup G \cup V$. The edges of H_3 are

$$\begin{aligned} E(H_3) = &\{\{w, v\} \mid w \in G \cup H, v \in V\} \cup \{\{2,4\},\{2',4'\}\} \cup \\ &\{\{0,0'\}\} \cup \{\{0,1\},\{1,2\},\{0,3\},\{2,3\}\} \cup \\ &\{\{0',1'\},\{1',2'\},\{0',3'\},\{2',3'\}\} \cup \\ &\{\{g_{11}, g_{12}, h_{11}, h_{12}, 4'\},\{g_{21}, g_{22}, h_{21}, h_{22}, 3\}, \\ &\{g_{11}, g_{21}, h_{11}, h_{21}, 4\},\{g_{12}, g_{22}, h_{12}, h_{22}, 3'\}\} \end{aligned}$$

The hypergraph is shown in Figure 6.2 with the edges $\{\{w, v\} \mid w \in G \cup H, v \in V\}$ omitted. We have $ghw(H_3) = 3$ and $hw(H_3) = 4$. We now show that indeed also $shw(H_3) = 3$. A witnessing decomposition is given in Figure 6.3. Note that, strictly speaking, only $mw(H_3) = 3$ is shown in [5] and, in general, mw is only a lower bound on ghw. However, it is easy to verify that $ghw(H_3) = 3$ holds by inspecting the winning strategy for 3 marshals in [5]. Moreover, the ghw-result is, of course, implicit when we show $shw(H_3) = 3$ next. Actually, the bags in the decomposition given in Figure 6.3 are precisely the χ-labels one would choose in a GHD of width 3. And the λ-labels are obtained from the (non-monotone) winning strategy for 3 marshals in [5].

To prove $shw(H_3) = 3$, it remains to verify that all the bags in Figure 6.3 are contained in $\mathsf{Soft}_{H_3,3}$. We see that $G \cup H$ are in all the bags and we, therefore, focus on the remaining part of the bags in our discussion. For the root, this is $\{3, 0', 0\}$. Here, the natural cover λ_1 consists of the two large horizontal edges in Figure 6.2 together with the edge $\{0, 0'\}$. The union $\bigcup \lambda_1$ contains the additional vertex $4'$. As λ_2 of Equation (6.1) we use the same two large edges plus $\{4', 2'\}$. As above, there is only one $[\lambda_2]$-edge component, which contains all vertices but $4'$. Note that, in contrast to the previous example, $4'$ in fact has high degree, but all of the edges that touch $4'$ are inside the separator.

Except for the bag $G \cup H \cup \{2, 4\}$, the bags always miss either vertex 4 or $4'$ from the "natural" covers, and the arguments are analogous to the ones above for the root bag. For the remaining bag, we consider the cover λ_1 consisting of the two vertical large edges plus the additional edge $\{2, 4\}$. Thus, we have the problematic vertex $3'$, which is contained in $\bigcup \lambda_1$ but not in the bag. Here, we observe a more complex scenario than before. As λ_2 take the two large horizontal edges plus $\{0', 1'\}$. This splits H_3 into two $[\lambda_2]$-components: one with vertices $G \cup H \cup \{0', 0, 1, 2, 3, 4\}$ and another with vertices $G \cup H \cup \{1', 2', 3', 4'\}$. The intersection of $\bigcup \lambda_1$ with the first component will produce the desired bag.

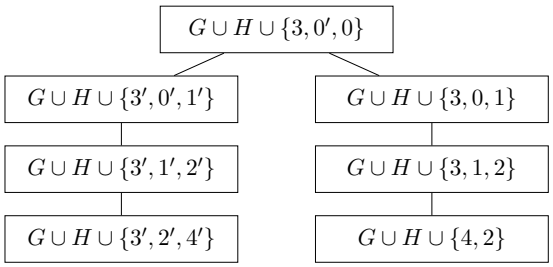


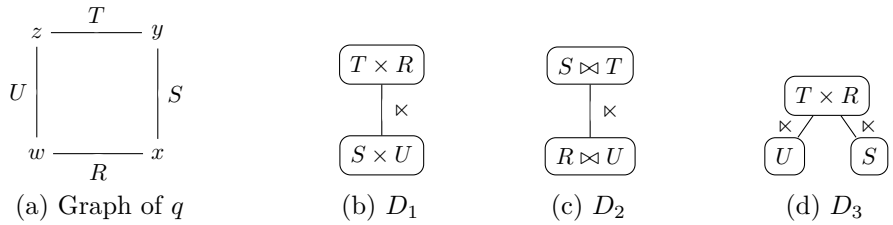
Figure 6.3: A soft hypertree decomposition of H_3 with width 3.

6.3 Constrained Hypertree Decompositions

Although hypertree decompositions and their generalisations have long been a central tool in identifying the asymptotic worst-case complexity of CQ evaluation, practical applications often demand more than simply a decomposition of low width. While width is the only relevant factor for the typically considered complexity upper bounds,

structural properties of the decomposition can critically influence computational efficiency in practice.

Example 6.3.1. *Consider the query $q = R(w,x) \wedge S(x,y) \wedge T(y,z) \wedge U(z,w)$, forming a 4-cycle. This query has multiple HDs of minimal width, but many of them are highly problematic for practical query evaluation. Some example computations resulting from various HDs of minimal width are illustrated below in (b)-(d).*



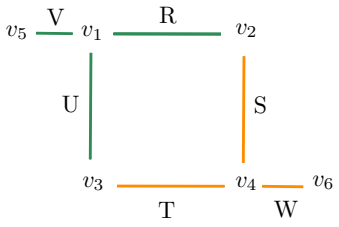
(a) Graph of q (b) D_1 (c) D_2 (d) D_3

Yannakakis' algorithm for decompositions D_1 and D_3 requires the computation of a Cartesian product (by covering w,x,y,z with two disjoint edges) of size $|T| \cdot |R|$ and $|S| \cdot |U|$, respectively. Under common practical circumstances, the joins $S \bowtie T$ and $R \bowtie U$ are much more efficient to compute.

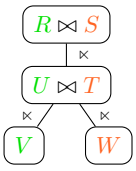
Example 6.3.2. *Consider the conjunctive query*

$$q = R(v_1,v_2) \wedge S(v_2,v_4) \wedge T(v_3,v_4) \wedge U(v_1,v_3) \wedge V(v_1,v_5) \wedge W(v_4,v_6).$$

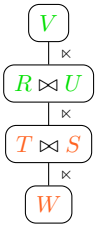
Assume furthermore, that we are in a distributed setting with vertical partitioning. In particular, relations R,U,V are on one node, whereas S,T,W are on another. The query hypergraph, together with the partitioning, is illustrated in Figure 6.5a. This query has multiple simple HDs of minimal width, with little natural reason to prefer one over the other. Some example computations resulting from various HDs of minimal width are illustrated below in (b)-(c).



(a) The query hypergraph H. Edge colours represent the partition of the respective relation.



(b) The computations for answering the query according to a natural HD of H.



(c) The computations for answering the query according to a different natural HD of H.

However, in a distributed setting where we want to minimise the communication between different nodes, we might prefer some HDs over others. In our example Figure 6.5c

might be preferable. Here, the bottom half of the decomposition is evaluated purely on the orange node, with only the result of $(T \bowtie S) \ltimes W$ being communicated to the other node. On the other hand, in Figure 6.5b two of the joins are across partitions, and the semi-join between them might add further communication depending on where it is best to compute the joins. Since HDs as in Figure 6.5c allow for simpler distributed query answering, some applications might only want HDs that are constrained to such HDs, where partitions are split up over disjoint subtrees of the decomposition. Naturally the full details of a practical scenario would require further details, e.g., whether $T \bowtie S$ is expected to be very large, but this simplified setting already illustrates the fundamental need for decompositions that follow certain constraints.

This issue is one of the simplest cases that highlights the necessity of imposing additional structural constraints on decompositions. By integrating such constraints, we can align decompositions more closely with practical considerations beyond worst-case complexity guarantees. We thus initiate the study of *constrained* decompositions. Specifically, for a constraint $\mathcal{C}$ and width measure *width*, we study $\mathcal{C}$-*width*(H), the least *width* over all decompositions that satisfy the constraint $\mathcal{C}$. In the first place, we thus study constrained *shw*. But we emphasise that the results in this section apply to any notion of decomposition and width that can be computed via CTDs.

To guide the following presentation, we first identify various interesting examples of constraints for a TD (T, B), that we believe might find use in applications:

Connected covers As discussed in Theorem 6.3.1, applications for query evaluation naturally want to avoid Cartesian products in the reduction from a hypertree decomposition to an acyclic query. This motivates the constraint ConCov, that holds exactly for those CTDs where every bag B_u has an edge cover λ_u of size $|\lambda_u| \leq k$, such that the edges in λ_u form a connected subhypergraph.

Shallow Cyclicity The *cyclicity depth* of a CTD for H is the least d such that the bag B_u of every node u at depth greater than d can be covered by a single edge of H. The constraint $\mathsf{ShallowCyc}_d$ is satisfied by a CTD, if it has cyclicity depth at most d. Intuitively, this constraint captures having a cyclic "core" to the query with acyclic parts attached to it. Such structure with low cyclicity depth can be naturally leveraged for efficient query answering by reducing the relations in the high-width nodes through semi-joins with the attached acyclic parts.

Partition Clustering As discussed in Theorem 6.3.2, in distributed scenarios, where relations are partitioned across the network, query evaluation using a decomposition could benefit substantially from being able to evaluate entire subtrees of the CTD at a single partition. We can enforce this through a constraint of the following form: Let $\rho : E(H) \to \mathit{Partitions}$ label each edge of the hypergraph with a partition. The constraint PartClust holds in a CTD (T, B), if there exists a function $f : V(T) \to \mathit{Partitions}$ such, that for every node u in T:

Algorithm 6.2: $(\mathcal{C}, \leq)$-Candidate Tree Decomposition of $\mathbf{S}$

```
      ⋮
5  repeat
6  |  foreach (B, C) ∈ blocks do
7  |  |  foreach X ∈ S \ {B} do
8  |  |  |  if X is a basis of (B, C) then
9  |  |  |  |  D_new ← Decomp(B, C, X)
10 |  |  |  |  if D_new ⊨ C and (basis(B, C) = ⊥ or
   |  |  |  |    D_new < Decomp(B, C, basis(B, C)) then basis(B, C) ← X
11 |  |  |  end
12 |  |  end
13 |  end
14 until no blocks changed
      ⋮
```

1. B_u has a candidate cover using edges with label $f(u)$.
2. For every $p \in \mathit{Partitions}$, the set of nodes u with $f(u) = p$ induces a subtree of T that is disjoint from the respective induced subtrees of all other partitions.

Introducing such constraints can, of course, increase the width. For example, for the 5-cycle C_5, it is easy to verify that ConCov-$ghw(C_5)$ = ConCov-$shw(C_5)$ = ConCov-$hw(C_5) = 3$ even though $hw(C_5) = 2$. However, in practice, adherence to such constraints can still be beneficial. In the C_5 example, the decomposition of width 2 forces a Cartesian product in the evaluation and is likely to be infeasible even on moderately sized data. Yet, using a ConCov decomposition of width 3 might even outperform a typical query plan of two-way joins executed by a standard relational DBMS. Connected decompositions of higher width are already used in certain homomorphism counting applications of hypertree decompositions [88, 17] for large graphs where computing the cross products for bags is prohibitive.

6.3.1 Tractable Constrained Decompositions

Here, we are interested in the question of when it is tractable to find decompositions satisfying certain constraints. The bottom-up process of Algorithm 6.1 iteratively builds tree-decompositions for some induced subhypergraph of H (recall that a satisfied block (B, C) corresponds to a TD for $H[B \cup C]$). We will refer to such tree decompositions as *partial tree decompositions* of H.

A *subtree constraint* $\mathcal{C}$ (or, simply, a constraint) is a Boolean property of partial tree decompositions of a given hypergraph H. We write $\mathcal{T} \models \mathcal{C}$ to say that the constraint holds true on the partial tree decomposition $\mathcal{T}$. A tree decomposition (T, B) of H satisfies $\mathcal{C}$ if $T_u \models \mathcal{C}$ for every node the partial tree decomposition induced by T_u.

However, deciding the existence of a tree decomposition satisfying $\mathcal{C}$ might require the prohibitively expensive enumeration of all possible decompositions for a given set of candidate bags. To establish tractability for a large number of constraints, we additionally consider *total quasiorderings of partial tree decompositions (toptds)*[1] $\leq$. A tree decomposition (T, B) is *globally minimal* w.r.t. $\leq$, if for every node u, there is no partial tree decomposition (T'_u, B') of $H[B(T_u)]$ with $(T'_u, B') < (T_u, B)$. We note that our notion of minimality is closely related to the notion of split-monotonicity introduced by Ravid et al. [135] to ensure that the cost of a TD cannot increase if a subtree T' of this TD is replaced by a subtree T'' of lower cost. Our goal here is to reduce the task of finding decompositions that satisfy certain constraints to the task of finding globally minimal decompositions for an appropriate toptd. To formalise when this is possible, we introduce the following property.

Definition 6.3.1. *We say that a toptd $\leq$ is* preference complete *for a subtree constraint $\mathcal{C}$ if the following holds. For every hypergraph H, if there exists a CTD of H for which $\mathcal{C}$ holds, then $\mathcal{C}$ holds for all globally minimal (w.r.t. $\leq$) CTDs of H.*

Example 6.3.3. *Consider the constraint* $\mathsf{ShallowCyc}_d$ *described above. We observe that the property is in a sense monotone over subtrees of the decomposition, i.e., the shallow cyclicity at node u cannot be lower than the maximum shallow cyclicity of the partial tree decompositions rooted at the children of u. Hence, we obtain a decomposition with the least shallow cyclicity if we cover the components below with their respective shallowest partial decompositions.*

If we thus consider the toptd $\leq_{\mathsf{ShallowCyc}_d}$ that simply orders partial tree decompositions according to their shallow cyclicity, a globally minimum CTD for $\leq_{\mathsf{ShallowCyc}_d}$ will be a CTD that achieves the least shallow cyclicity. In particular, we get that all the globally minimum CTDs will, by definition, have the same shallow cyclicity. Hence, if any of them satisfies $\mathsf{ShallowCyc}_d$*, then all of them do. Therefore, this toptd is preference complete for* $\mathsf{ShallowCyc}_d$*.*

Subtree constraints – even with the additional condition of preference completeness – still include a wide range of constraints. For example, all three example constraints mentioned above are preference complete. Informally, for shallow cyclicity, those partial TDs that become acyclic from lower depth are preferred. For partition clustering, we prefer the root node of the partial TD to be in the same partition as one of the children over introducing a new partition.

However, efficient search for constrained decompositions is not the only use of toptds. Using the same algorithmic framework, one can see them as a way to order decompositions by preference as long as the respective toptd $\leq$ is *strongly monotone*: that is, a partial tree decomposition (T, B) is globally minimal only if, for each child c of the root, (T_c, B) is globally minimal. A typical example for strongly monotone toptd are cost functions for

[1] Recall that a total quasiordering of X is a reflexive, transitive, and total relation on X^2.

the estimated cost to evaluate a database query corresponding to the respective subtree. The cost of solving a query with a tree decomposition is made up of the costs for the individual subqueries of the child subtrees, plus some estimated cost of combining them, strong monotonicity is a natural simplifying assumption for this setting. For instance, consider a quasiordering of partial subtrees $\leq_{cost}$ that orders partial decompositions by such a cost estimate. In combination with the constraint ConCov, (ConCov, $\leq_{cost}$) forms a preference complete subtree constraint, such that a satisfying TD is a globally minimal cost tree decomposition where every bag has a connected edge cover of size at most k. Thus, optimisation of tree decompositions for strongly monotone toptds is simply a special case of preference complete subtree constraints.

Our main goal in introducing the above concepts is the general study of the complexity of incorporating constraints into the computation of CTDs. We define the $(\mathcal{C}, \leq)$-CandidateTD problem as the problem of deciding for a given hypergraph H and set S of candidate bags whether there exists a CompNF CTD that satisfies $\mathcal{C}$ and is minimal for $\leq$. We want to identify tractable fragments of $(\mathcal{C}, \leq_{\mathcal{C}})$-CandidateTD. Let us call a constraint and toptd *tractable* if one can decide in polynomial time w.r.t. the size of the original hypergraph H and the set S of candidate bags both, whether $\mathcal{C}$ holds for a partial tree decomposition of H, and whether $(T', B') <_{\mathcal{C}} (T, B)$ holds for partial tree decompositions of H.

Theorem 6.3.4. *Let $\mathcal{C}, \leq$ be a tractable constraint and toptd such that $\leq$ is preference complete for $\mathcal{C}$. Then $(\mathcal{C}, \leq)$-CandidateTD $\in$ PTIME.*

Proof Sketch. We obtain a polynomial time algorithm for $(\mathcal{C}, \leq)$-CandidateTD by modifying the main loop of Algorithm 6.1 as shown in Algorithm 6.2 (everything outside of the repeat loop remains unchanged). For a block (S, C) with basis X in Algorithm 6.2, the *basis* property of every block induces a (unique) tree decomposition for $S \cup C$, denoted as $Decomp(S, C, X)$. Where Algorithm 6.1 used dynamic programming to simply check for a possible way to satisfy the root block $(\emptyset, V(H))$. We instead use dynamic programming to find the preferred way, that is a basis that induces a globally minimal partial tree decomposition, to satisfy blocks. It is straightforward to verify the correctness of Algorithm 6.2. The polynomial time upper bound follows directly from the bound for Algorithm 6.1 and our tractability assumptions on $(\mathcal{C}, \leq_{\mathcal{C}})$. □

Our analysis here brings us back to one of the initially raised benefits of soft hypertree width: algorithmic flexibility. Our analysis, and Theorem 6.3.4 in particular, applies to any setting where a width measure can be effectively expressed in terms of CompNF candidate tree decompositions.

Corollary 6.3.5. *Let $\mathcal{C}$ be a subtree constraint, let $\leq$ be a preference complete toptd for $\mathcal{C}$, and suppose that $\mathcal{C}, \leq$ are tractable. Let k be a non-negative integer. Then deciding $\mathcal{C}$-shw$(H) \leq k$ is feasible in polynomial time.*

Using CTDs, Gottlob et al. [60, 59] identified large fragments for which checking $ghw \leq k$ or $fhw \leq k$ is tractable. Hence, results analogous to Theorem 6.3.5 follow immediately also for tractable fragments of generalised and fractional hypertree decompositions.

6.3.2 Constraints in Other Approaches for Computing Decompositions

Bag-level constraints like ConCov are easy to enforce using existing combinatorial algorithms such as `det-k-decomp` [66] and `new-det-k-decomp` [54], which construct an HD top-down by combining up to k edges into λ-labels. Enforcing the ConCov is trivial in this case. However, enforcing more global constraints like PartClust while maintaining tractability remains unclear, as does integrating a cost function. These algorithms critically rely on caching for pairs of a bag B and vertex set C, whether there is an HD of $H[C]$ rooted at B. With constraints that apply to a larger portion of the decomposition, the caching mechanism no longer works. Other algorithms are also unsuitable: `log-k-decomp` [58], the fastest HD algorithm in practice, applies a divide-and-conquer strategy, splitting the hypergraph until a base case is reached, but never analyzes larger sections of the decomposition. The HtdSMT solver by Schidler and Szeider [137, 138] minimises HD width via SMT solving. It is unclear how to state constraints over these encodings. Furthermore, due to the reliance on SAT/SMT solvers, the method is not well suited for a theoretical analysis of tractable constrained decomposition methods.

The algorithm that comes closest to our algorithm for constructing a constrained decomposition is the `opt-k-decomp` algorithm from [136], which constructs a "weighted HD" of minimal cost up to a given width. The cost of an HD is defined by a function that assigns a cost to each node and each edge in the HD. The natural cost function assigns the cost of the join computation of the relations in λ_u to each node u and the cost of the (semi-)join between the bags at a node and its parent node to the corresponding edge. We also consider this type of cost function (and the goal to minimise the cost) as an important special case of a preference relation in our framework. However, our CTD-based construction of constrained decomposition allows for a greater variety of preferences and constraints, and the simplicity of Algorithm 6.2 facilitates a straightforward complexity analysis. Moreover, `opt-k-decomp` has been specifically designed for HDs. This is in contrast to our framework, which guarantees tractability for any combination of decompositions with tractable, preference complete subtree constraints as long as the set of candidate sets is polynomially bounded. As mentioned above, this is, for instance, the case for the tractable fragments of generalised and fractional HDs identified in [60, 59].

6.4 Implementation

This section gives technical details on the tools that we used and implemented. It will also detail the cost functions, which extract statistics from the database to assign a cost estimate to a CTD, where a lower value indicates lower estimated runtime. Our tools developed in this chapter are partially derived from the Scala code of the

Spark SQL integration in Chapter 4. Our source code is available publicly: `https://github.com/cem-okulmus/softhw-pods25`.

6.4.1 Implementation Overview

Our system evaluates SQL queries over a database, but does so by way of first finding an optimal decomposition, then using the decomposition to produce a *rewriting* - a series of SQL queries which together produce the semantically equivalent result as the input query – and then runs this rewriting on the target DBMS. This procedure is meant to guide the DBMS by exploiting the structure present in the optimal decomposition. A graphical overview of the system, which provides an end-to-end implementation of the process of finding an optimal decomposition for a SQL query over a database, up to running the rewritten query on the DBMS, is given in Figure 6.6. For simplicity, we assume here that only Postgres is used.

The system consists of two major components – a Python library providing an interface to the user, which handles the search for the best decompositions, and a Scala component which connects to the DBMS and makes use of Apache Calcite to parse the SQL query and extract its hypergraph structure, as well as extracting from the DBMS node cost information (which we will detail further below) and generate the rewritings.

The Python library – referred to as *QueryRewriterPython* – acts as a proxy to the DBMS, and is thus initialised like a standard DBMS connection. The library is capable of returning not just the optimal rewriting, but can provide the top-n best rewritings. In practice, usually, the single best rewriting would be chosen. However, retrieving the top n rewritings comes at an insignificant extra cost and may be beneficial when the DBMS cost estimates are unreliable. As seen in Figure 6.6, on start up QueryRewriterPython starts *QueryRewriterScala* as a sub-process. QueryRewriterPython communicates with it via RPC calls using Py4J. QueryRewriterScala obtains the JDBC connection details from QueryRewriterPython, and connects to the DBMS using Apache Calcite, in order to be able to access the schema information required later. The input SQL query is then passed to QueryRewriterScala. Using Apache Calcite, and the schema information from the database, the input SQL query is parsed and converted into a logical query plan. After applying simple optimiser rules in order to obtain a convenient representation, the join structure of the plan is extracted and used to construct a hypergraph. Subtrees in the logical representation of the query, which are the inputs to joins, such as table scans followed by projection and filters, are kept track of, and form the hyperedges of this hypergraph. Later, when creating the rewriting, the corresponding SQL queries for these subtrees is used in the generation of the leaf node VIEW expressions. After retrieving the hypergraph, the QueryRewriterPython enumerates the possible covers, i.e., hypertree nodes, whose size is based on the width parameter k. The list of nodes is sent to the QueryRewriterScala, where the costs of each of these nodes are estimated by running EXPLAIN statements on the database. In order to reduce the overhead, as the number of possible nodes can quickly become large, this is done in parallel using multiple connections to the DBMS. Similarly, the costs of the semijoins between nodes can also

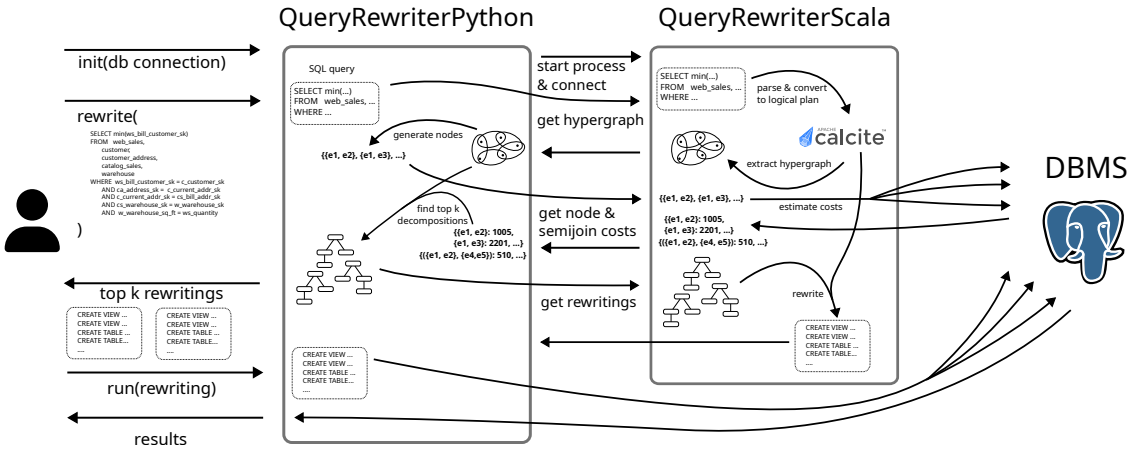


Figure 6.6: Overview of the components of the query rewriting system

be estimated. We will detail the two used cost functions in the next subsection. Once all the cost functions have been computed, the QueryRewriterPython follows an algorithm in the vein of Algorithm 6.2, in order to find the best or the top-n best decompositions. Finally, the decompositions are passed to the QueryRewriterScala in order to generate the rewritings.

6.4.2 Cost Functions

In this section, we describe the cost functions implemented which will be used for our experiments in Section 6.5.

A Cost Function Based on DBMS-Estimates

To estimate the costs of the decompositions, we make use of Postgres' cost estimates, as they are internally used by the system for finding a good query plan. Postgres estimates the costs of a query plan in abstract units based on assumptions about lower-level costs in the system, such as disk I/O, or CPU operations. By making use of EXPLAIN statements, it is possible to retrieve the total cost of a plan estimated by the DBMS.

When estimating the costs of bags in the decomposition, we construct the join query corresponding to the bag, and let Postgres estimate the cost of this query in an EXPLAIN statement.

$$cost(u) := \begin{cases} C(\pi_{B_u}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)) & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases} \tag{6.2}$$

where $C(q)$ corresponds to the costs estimated by Postgres for the query q.

To estimate the costs of a subtree, we also consider the costs of computing the bottom-up semijoins. Because Postgres includes the costs of scanning the relations which are semijoined, and any joins inside the bags, in the total costs, we have to subtract these costs from the semijoin-costs. Due to noisy estimates, we have to avoid the total cost becoming negative, and thus set a minimum cost.

$$cost(T_p) := cost(p) + \sum_{i=1}^{k} (cost(T_{c_i}) + min(C(J_p \ltimes J_{c_i}) - C(J_p) - C(J_{c_i}), 1)) \tag{6.3}$$

A Cost Function Based on Actual Cardinalities

Because we observe problematic unreliability of cost estimates from the DBMS, we also consider an idealised cost function, that is omniscient about bag sizes and uses these to estimate cost of individual operations. This has the weakness of not taking physical information, such as whether relations are already in memory (or even fit wholly in memory) or specific implementation behaviour (which we surprisingly observe to be highly relevant in our experiments). Nonetheless, we find it instructive to compare these costs to the estimate based cost function to correct for cases where estimates are wildly inaccurate.

First, the cost for bags. We will assume we know the cardinality of the bag (simulating a good query planner) and combine this with the size of the relations that make up the join. The idea is simple, to compute the join we need to at least scan every relation that makes up the join (that is, linear effort in the size of the relations). And then we also need to create the new relation, which takes effort linear in the size of the join result. We thus get for a node u of a decomposition, whose bag is created by cover $R_1, \ldots, R_n$:

$$cost(u) := \begin{cases} |J_u| + |\sum_{i=1}^{n} |R_i| \cdot \log |R_i| & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases} \tag{6.4}$$

where $J_u = \pi_{B_u}(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n)$, i.e., table for the node.

The cost for a subtree then builds on this, together with some estimate of how effective the semi-joins from the subtrees will be. We will estimate this very simply, by seeing how many non primary-key attributes of the parent appear in a subtree. The point being that if something is a FK in the parent, we assume that every semi-join will do nothing. If something is not a PK/FK relationship in this direction, then we assume some multiplicative reduction in tuples for each such attribute. For every node p in a rooted decomposition, we thus define $ReduceAttrs(p)$ as the set containing every attribute A in B_p, for which A appears in a subtree rooted at a child of p, where it does not come from a relation where A is the primary key.

With this in hand we then estimate the size of the node after the up-phase semi-joins have reached it:

$$ReducedSz(u) := \begin{cases} 0 & \text{if } ReducedSz(c) = 0 \text{ for any child } c \text{ of } u \\ \frac{|J_u|}{1+|ReduceAttrs(u)|} & \text{otherwise} \end{cases} \tag{6.5}$$

Importantly, if relation C is empty, PostgreSQL never scans U in the semi-join $U \ltimes C$, to capture this define the following for nodes u.

$$ScanCost(u) := \begin{cases} 0 & \min_{c \in children(u)} ReducedSz(c) = 0 \\ |J_u| \cdot \log |J_u| & \text{otherwise} \end{cases}$$

The cost for a subtree T_p rooted at p, with children $c_1, \ldots, c_k$ is then as follows:

$$cost(T_p) := cost(p) + ScanCost(p) + \sum_{i=1}^{k} (cost(T_{c_i}) + ReducedSz(c_i) \cdot \log ReducedSz(c_i)) \tag{6.6}$$

which corresponds to computing the bag for p, computing the subtrees for each child, and then finally also using $ReducedSz(c_i)$ as a proxy for the cost of the semi-join $p \ltimes c_i$. From experiments, it seems like PostgreSQL actually does the initial loop over the right-hand side of the semi-join. So when c_i is empty, it stops instantly, but when p is empty it does a full scan of c_i. This is better reflected by simply taking the reduced child size as the cost.

6.5 Experimental Results

While the focus in this chapter is primarily on the theory of tractable decompositions of hypergraphs, the ultimate motivation of this work is drawn from applications to database query evaluation. We therefore include a focused experimental analysis of the practical effect of constraints and optimisation on candidate tree decompositions. The aim of our experiments is to gain insights into the effects of constraints and costs on candidate tree decompositions in practical settings. We perform experiments with cyclic queries over standard benchmarks (TPC-DS [133], LSQB [116] as well as queries over the Hetionet Biomedical Knowledge Graph [80] from a recent cardinality estimation benchmark [23]. In all experiments, we first compute candidate tree decompositions as in Algorithm 6.2 for the candidate bags as in Definition 6.2.1, i.e., we compute constrained soft hypertree decompositions. We then perform a Yannakakis rewriting as in Chapter 4, to execute Yannakakis' algorithm for these decompositions on standard relational DBMSs. Our implementation is publicly available at `https://github.com/cem-okulmus/softhw-pods25`. We note that a related analysis specifically for hypertree width was performed in the past by Scarcello et al. [136].

Experimental Setup. The tests were run on a test server with an AMD EPYC-Milan Processor with 16 cores, run at a max. 2GHz clock speed, with 128 GB RAM. The

test data and PostgreSQL database was stored and running off of a 500GB SSD. The operating system was Ubuntu 22.04.2 LTS, running Linux kernel with version 5.15.0-75-generic. We executed the tests using a JupyterLab Notebook, which we make available in `https://github.com/cem-okulmus/softhw-pods25`. This also includes the source code of the Scala library that handles the tasks of parsing the input query, producing the hypergraph and extracting costs from the target DBMS. We also include all experimental data collected in this repository.

Benchmarks. We consider three benchmarks:

- TPC-DS [149], which is already known from the experiments in Chapter 5. For the following experiments, we use the scaling factor 10.
- Queries already presented in [23] over the hetionet dataset [80].
- LSQB [117], which is also known from Chapter 5. We use the scaling factor 10.

Queries. Since the focus is on queries that exhibit high join costs, we opted to manually construct such challenging yet still practical examples, due to the fact that many benchmarks, such as TPC-DS opt to focus on cases where joins are always along primary or foreign keys, and in case of foreign keys only to join them with the respect table they are a key of. This means that joins do not produce significantly more rows than are present in the involved tables: to see this, we just observe that a join over a key can, by the uniqueness condition, only find exactly one match (and by referential integrity, must always succeed). While this scenario makes sense when users have carefully prepared databases with a clear schema, we believe and indeed users reports suggest, that queries with a large number of tables with very high intermediate join sizes occur in practice. So to get any useful insights into our framework's applicability, we forgo the default provided queries and pick a number of handcrafted "tough" join queries, putting more focus on their evaluation complexity and less on their semantics. A notable exception here are the queries from Hetionet, which we directly sourced together with data from [23], which are used as-is, since they already exhibit complex, cyclic join queries.

This is not meant to be a thorough, systematic analysis, which would involve a very large number of cyclic queries, which are currently rare in standard benchmarks, tested across multiple systems. That goes beyond the scope of this work, which only aims to show the *practical potential* of our framework. Implementing it in a system which enables reliable, robust performance across all kinds of queries is left as future work, once its potential has been proven.

We choose 6 queries: q^{ds} from TPC-DS, q_1^{hto} to q_4^{hto} from hetionet, and q^{lb} from LSQB. In Table 6.1 we provide details on these queries, including their *shw* and other measures, as explained in the caption.

Query	ConCov-*shw*	$\lvert H\rvert$	$\mathsf{Soft}_{H,k}$	$\mathsf{ConCov\text{-}Soft}_{H,k}$	Time to produce top-10 best TDs
q^{ds}	2	5	9	8	7.67 ms
q_1^{hto}	2	7	25	16	27.87 ms
q_2^{hto}	2	7	25	16	26.58 ms
q_3^{hto}	2	4	9	8	3.26 ms
q_4^{hto}	2	6	17	12	23.26 ms
q^{lb}	3	6	17	15	26.42 ms

Table 6.1: For each of the 6 queries, we list its connected SoftHW, the size of its hypergraph, the size of its soft set and the size of its connected soft set. In addition, we report on the time it took to produce the top-10 best TDs, according to the cost measure from Section 6.4.2.

Results. We study the effect of optimising candidate tree decompositions that adhere to the ConCov constraint over two cost functions, choosing the TPC-DS query q^{ds}. The first cost function is based on estimated costs of joins and semi-joins by the DBMS (PostgreSQL) itself, the second is derived from the actual cardinalities of relations and joins (see Section 6.4.2) for details). The results of these experiments are summarised in Figure 6.7. These findings indicate that while certain decompositions can cut the execution time by more than half, others can be nearly ten times slower. This stark variation underscores the critical need for informed selection strategies when deploying decompositions for real-world database workloads.

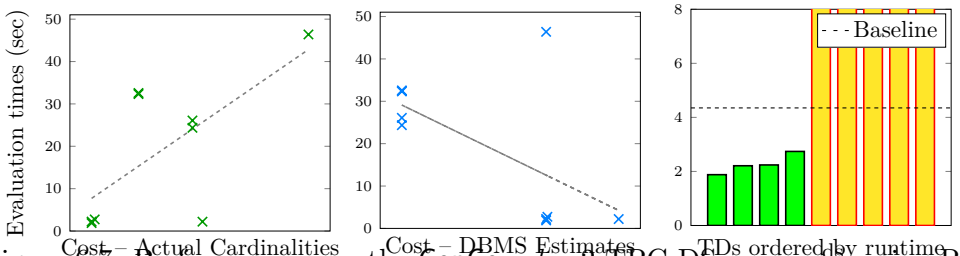


Figure 6.7: Performance over the ConCov-*shw* 2 TPC-DS query q^{ds}, using PostgreSQL as a backend.

A second dimension of interest is the efficacy of the cost function. A priori, we expect the costs derived directly from the DBMS cost estimates to provide the stronger correlation between cost and time. However, we observe that the cost estimates of the DBMS are sometimes very unreliable, especially when it comes to cyclic queries[2]. Clearly, this makes it even harder to find good decompositions, as a cost function would ideally be based on good estimates. The comparison of cost functions in Figure 6.7 highlights

[2]This is not particularly surprising and remains a widely studied topic of database research (see e.g., [41]).

this issue. There, we use as cost function the actual cardinalities as a proxy for good DBMS estimates. We see that the cost thus assigned to decompositions indeed neatly corresponds to query performance, whereas the cost using DBMS estimates inversely correlates to query performance.

To expand our analysis, we evaluated two graph queries over Hetionet. In Figure 6.8. we report on the 10 cheapest decompositions of width 2, for the queries . On these queries, both cost functions perform very similarly; we report costs based on DBMS estimates here. We still see a noticeable difference between decompositions, but more importantly, all of them are multiple times faster than the standard execution of the query in PostgreSQL. It turns out that connected covers alone are critical. In the right-most chart of Figure 6.8, we show the average time of executing the queries for 10 randomly chosen decompositions of width 2 with and without the ConCov constraint enforced. We see that the constraint alone is already sufficient to achieve significant improvements over standard execution in relational DBMSs.

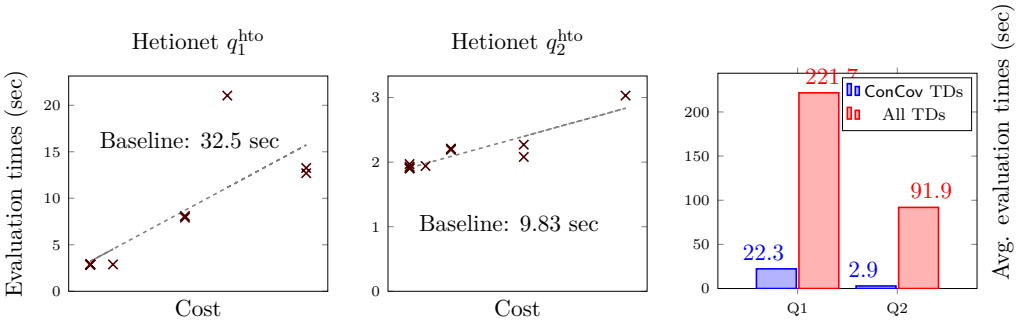


Figure 6.8: Performance over two ConCov-*shw* 2 Hetionet queries (q_1^{hto} and q_2^{hto}) using PostgreSQL as a backend.

Further details on the cardinality estimates of the queries in Figure 6.8 are provided in Figures 6.9 and 6.10. Additionally, we also report the results from q_3^{hto} and q_4^{hto} in Figure 6.11 and 6.12. In Figure 6.13, we provide the results for the LSQB query q^{lb}.

Our implementation computes candidate tree decompositions closely following Algorithm 6.2. We find that the set of candidate bags in real world queries is very small, especially compared to the theoretical bounds on the set $\mathsf{Soft}_{H,k}$. The TPC-DS query from Figure 6.7 has only 9 elements in $\mathsf{Soft}_{H,2}$ (one of which does not satisfy ConCov). The two queries in Figure 6.8 both have 25 candidate bags, 16 of which satisfy ConCov. Accordingly, it takes only a few milliseconds to enumerate all decompositions ranked by cost in these examples.

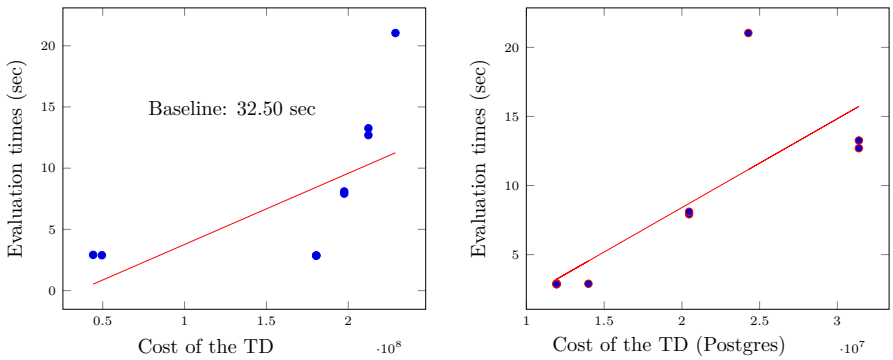


Figure 6.9: Performance over q_{hto} over the Hetionet using PostgreSQL as a backend.

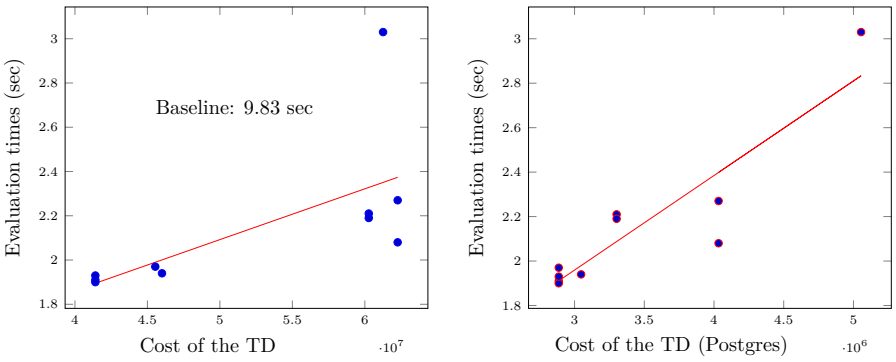


Figure 6.10: Performance over $q2_{\text{hto}}$ over the Hetionet using PostgreSQL as a backend.

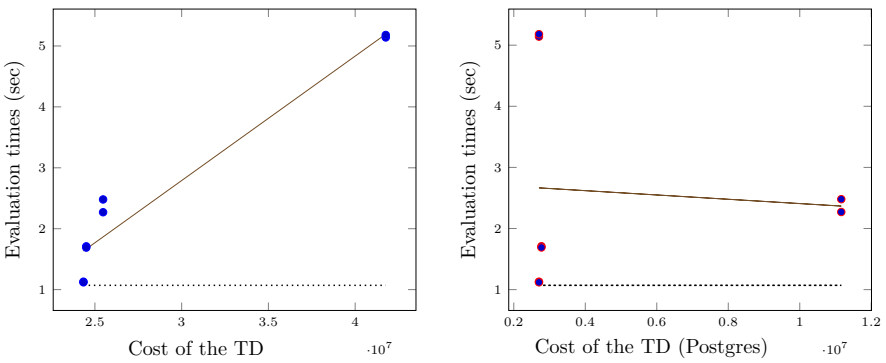


Figure 6.11: Performance over $q3_{\text{hto}}$ over the Hetionet using PostgreSQL as a backend.

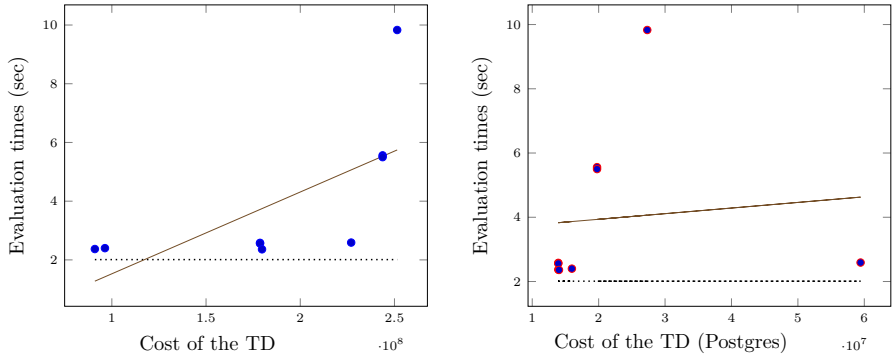


Figure 6.12: Performance over $q4_{\text{hto}}$ over the Hetionet using PostgreSQL as a backend.

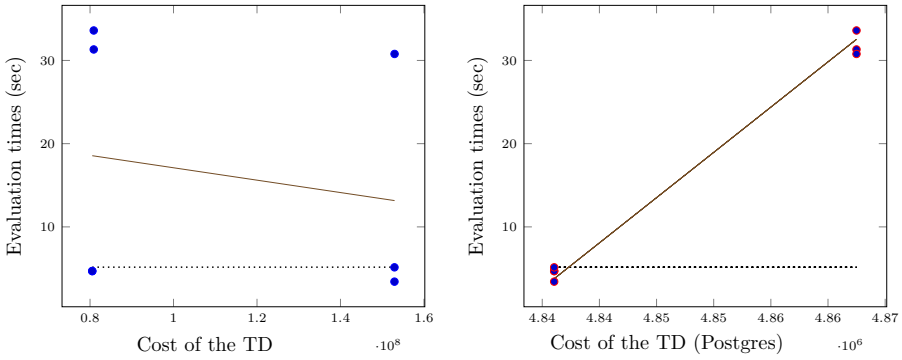


Figure 6.13: Performance over q_{lb} over the LSQB Benchmark using PostgreSQL as a backend.

6.6 Summary

In this chapter, we have introduced the concept of soft hypertree decompositions (soft HDs) and the associated measure of soft hypertree width (*shw*). Despite avoiding the special condition of HDs, we retained the tractability of deciding whether a given CQ has width at most k. At the same time, *shw* is never greater than *hw* and it may allow for strictly smaller widths. Most importantly, it provides more algorithmic flexibility. Building on the framework of candidate tree decompositions (CTDs), we have demonstrated how to incorporate diverse constraints and preferences into the decomposition process, enabling more specialised decompositions that take application-specific concerns beyond width into account. We have implemented an end-to-end pipeline, enabling the use of these techniques to rewrite queries automatically. By performing experimental evaluation, we confirm that this approach improves the quality of decompositions and thus leads to performance gains in practice.

CHAPTER 7

Conclusion

In this final chapter, we summarise the main contributions of this thesis. Finally, we discuss potential future research directions identified in the course of our work.

7.1 Summary of Contributions

We began our work in Chapter 4 by exploring the effectiveness of Yannakakis-style query evaluation in mainstream relational DBMSs. In the course of this investigation, we have introduced a class of queries – 0MA queries – which are particularly well suited for structure-guided query processing. We presented a rewriting-based approach towards Yannakakis-style query execution. Our experiments on multiple DBMSs indicated significant potential for speeding up join processing on hard instances, while at the same time showing that there are major hurdles towards practicability in general. Addressing the newly identified challenge that rewriting-based Yannakakis-style query execution delivers performance gains only in some cases but not in all cases, we framed the choice between Yannakakis-style optimisation and conventional DBMS evaluation as an algorithm selection problem. We developed a decision procedure, which, across multiple popular RDBMSs, delivers substantial performance improvements.

In Chapter 5 we approached the challenge of broadening the class of queries efficiently answerable without materialising join results while finally bringing an implementation into the core of a query processing system. To extend the 0MA class identified in Chapter 4 towards more practical aggregate queries, we first weakened the restrictions by eliminating the set-safety condition, thereby introducing the class of guarded aggregate queries. Furthermore, we extended the class even further by weakening the notion of guardedness to piecewise-guardedness, obtaining the class of piecewise-guarded aggregate queries. We performed an integration into the query optimiser of Spark SQL, allowing for the efficient execution of piecewise-guarded queries. Through performance evaluation over 5 benchmarks, we confirmed a wide applicability of the optimisations as well as

significant performance improvements without introducing overhead. Finally, we made significant steps towards covering arbitrary unguarded ACQs where materialisation of intermediate results cannot be avoided completely. By introducing the new physical operator GroupAggJoin and integrating it into the optimisation, we can avoid large amounts of materialisation. We introduced a new benchmark for unguarded aggregate queries and experimentally confirmed the effectiveness of this approach on unguarded queries. After observing that Yannakakis-style query processing can be successfully applied to vast amounts of ACQs, we shifted our attention towards the hard class of cyclic queries in Chapter 6. We have introduced the concept of soft hypertree decompositions (soft HDs) and the associated measure of soft hypertree width (*shw*). Despite avoiding the special condition of HDs, we retained the tractability of deciding whether a given CQ has width at most k. At the same time, *shw* is never greater than *hw* and it may allow for strictly smaller widths. Most importantly, it provides more algorithmic flexibility. Building on the framework of candidate tree decompositions (CTDs), we have demonstrated how to incorporate diverse constraints and preferences into the decomposition process, enabling more specialised decompositions that take application-specific concerns beyond width into account. We have implemented an end-to-end pipeline making these optimisations work seamlessly with standard DBMS. The experimental evaluation confirms that this approach can yield significant performance gains in practice, without introducing new computational bottlenecks.

7.2 Future Work

The work of this thesis raises several interesting future lines of research.

In our work in Chapter 5, we extended the implementation to unguarded ACQs via the GroupAggJoin operator, which still requires us to materialise some joins. By introducing the new benchmark JOBUnguarded, and evaluating the implementation on it, we noticed that queries with up to ≈ 5 unguarded attributes can be reliably optimised via this approach. However, beyond this threshold, while performance is not degraded, there is no strong outperformance. An interesting topic for future research would be to investigate more closely in which cases this approach is likely to outperform the baseline. This might again lead to similar ideas as in Chapter 4, as cost estimation could help make this decision.

Currently, unguarded aggregate expressions, such as `SUM(a*b)`, are handled by considering them in the same way as group by conditions, such as `GROUP BY a, b`. While still performing well, this approach is not the best we could do. For aggregations over attributes which come together at one node in the tree, there is no need to propagate them all the way up to the root. Therefore, it would be an interesting optimisation to perform the aggregations earlier.

Executing ACQs along a join tree as opposed to performing a sequence of binary joins could have great benefits for distributed query processing, where data transfer is expensive. Unfortunately, Spark SQL itself is not very effective at distributed joins.

Possibly, Yannakakis-style execution could be integrated more effectively into other distributed query engines or possibly parallelized implementations on a single machine.

Going beyond purely conjunctive queries, benchmarks often contain unions of conjunctive queries, inequality conditions, or other non-equality comparisons such as range conditions. It is an interesting topic for research to extend Yannakakis-style query execution to these cases.

The logical next step for the implementation introduced in Chapter 6 is to better explore cost measures and constraints. A further valuable extension would be to combine the techniques of Chapter 6 with an implementation into a system such as the extension of Spark SQL introduced in Chapter 5. This would remove the overhead of the external rewriting. Furthermore, a combination with machine-learning based algorithm selection as applied in Chapter 4 could possibly also be worthwhile.

Overview of Generative AI Tools Used

ChatGPT (GPT-5 and older versions) and Claude (sonnet 4.5 and older versions) were used for LaTeX questions, table generation, minor suggestions for text rephrasing (avoiding repetitions, suggesting synonyms, etc). Claude Code was used for data analysis, data transformation, and for generating code to create the dataset of unguarded queries.

Bibliography

[1] MusicBrainz - The Open Music Encyclopedia. https://musicbrainz.org/, 2022. [Online; accessed 24-July-2022].

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

[3] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addison-Wesley, 1995.

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

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

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

[7] Afrati, F. N., Joglekar, M. R., Ré, C., Salihoglu, S., and Ullman, J. D. GYM: A multiround distributed join algorithm. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy* (2017), M. Benedikt and G. Orsi, Eds., vol. 68 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 4:1–4:18.

[8] Akdere, M., Çetintemel, U., Riondato, M., Upfal, E., and Zdonik, S. B. Learning-based query performance modeling and prediction. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012* (2012), A. Kementsietsidis and M. A. V. Salles, Eds., IEEE Computer Society, pp. 390–401.

[9] Aken, D. V., Pavlo, A., Gordon, G. J., and Zhang, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), ACM, pp. 1009–1024.

[10] Angles, R., Antal, J. B., Averbuch, A., Boncz, P. A., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Larriba-Pey, J. L., Martínez-Bazan, N., Marton, J., Paradies, M., Pham, M., Prat-Pérez, A., Spasic, M., Steer, B. A., Szárnyas, G., and Waudby, J. The LDBC social network benchmark. *CoRR abs/2001.02299* (2020).

[11] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. Spark SQL: relational data processing in spark. In *Proceedings SIGMOD* (2015), ACM, pp. 1383–1394.

[12] Arnborg, S., Corneil, D. G., and Proskurowski, A. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods 8*, 2 (1987), 277–284.

[13] Bagan, G., Durand, A., and Grandjean, E. On acyclic conjunctive queries and constant delay enumeration. In *Proceedings CSL* (2007), vol. 4646 of *LNCS*, Springer, pp. 208–222.

[14] Bakibayev, N., Kociský, T., Olteanu, D., and Zavodny, J. Aggregation and ordering in factorised databases. *Proc. VLDB Endow. 6*, 14 (2013), 1990–2001.

[15] Baldacci, L., and Golfarelli, M. A cost model for SPARK SQL. *IEEE Trans. Knowl. Data Eng. 31*, 5 (2019), 819–832.

[16] Bao, L., Jin, E., Bronstein, M., İsmail İlkan Ceylan, and Lanzinger, M. Homomorphism counts as structural encodings for graph learning, 2024.

[17] Bao, L., Jin, E., Bronstein, M. M., Ceylan, I. I., and Lanzinger, M. Homomorphism counts as structural encodings for graph learning. In *Proceedings ICLR* (2025), OpenReview.net.

[18] Barceló, P., Geerts, F., Reutter, J. L., and Ryschkov, M. Graph neural networks with local graph parameters. In *Proceedings NeurIPS* (2021), pp. 25280–25293.

[19] Batista, L. O., de Silva, G. A., Araújo, V. S., Araújo, V. J. S., Rezende, T. S., Guimarães, A. J., and de Campos Souza, P. V. Fuzzy neural networks to create an expert system for detecting attacks by SQL injection. *CoRR abs/1901.02868* (2019).

[20] Bekkers, L., Neven, F., Vansummeren, S., and Wang, Y. R. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *CoRR abs/2411.04042* (2024).

[21] Bekkers, L., Neven, F., Vansummeren, S., and Wang, Y. R. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow. 18*, 8 (2025), 2413–2426.

[22] Bhaskar, R., Laxman, S., Smith, A. D., and Thakurta, A. Discovering frequent patterns in sensitive data. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, July 25-28, 2010* (2010), B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, Eds., ACM, pp. 503–512.

[23] Birler, A., Kemper, A., and Neumann, T. Robust join processing with diamond hardened joins. *Proc. VLDB Endow. 17*, 11 (2024), 3215–3228.

[24] Böhm, D. To rewrite or not to rewrite: Decision making in query optimization of sql queries. Master's thesis, Technische Universität Wien, 2024.

[25] Bonifati, A., Martens, W., and Timm, T. An analytical study of large SPARQL query logs. *VLDB J. 29*, 2-3 (2020), 655–679.

[26] Borgs, C., Chayes, J. T., Lovász, L., Sós, V. T., and Vesztergombi, K. Counting graph homomorphisms. *Topics in Discrete Mathematics* (2006), 315–371.

[27] Bouchitté, V., and Todinca, I. Minimal triangulations for graphs with "few" minimal separators. In *Proc. ESA* (1998), vol. 1461 of *Lecture Notes in Computer Science*, Springer, pp. 344–355.

[28] Bouchitté, V., and Todinca, I. Treewidth and minimum fill-in of weakly triangulated graphs. In *Proc. STACS* (1999), vol. 1563 of *Lecture Notes in Computer Science*, Springer, pp. 197–206.

[29] Bouchitté, V., and Todinca, I. Listing all potential maximal cliques of a graph. In *Proc. STACS* (2000), vol. 1770 of *Lecture Notes in Computer Science*, Springer, pp. 503–515.

[30] Bouchitté, V., and Todinca, I. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM J. Comput. 31*, 1 (2001), 212–232.

[31] Bouchitté, V., and Todinca, I. Listing all potential maximal cliques of a graph. *Theor. Comput. Sci. 276*, 1-2 (2002), 17–32.

[32] Breiman, L. *Classification and regression trees*. Routledge, 2017.

[33] Carmeli, N., and Kröll, M. Enumeration complexity of conjunctive queries with functional dependencies. *Theory Comput. Syst. 64*, 5 (2020), 828–860.

[34] Carmeli, N., and Kröll, M. On the enumeration complexity of unions of conjunctive queries. *ACM Trans. Database Syst. 46*, 2 (2021), 5:1–5:41.

[35] Carmeli, N., Tziavelis, N., Gatterbauer, W., Kimelfeld, B., and Riedewald, M. Tractable orders for direct access to ranked answers of conjunctive queries. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021* (2021), L. Libkin, R. Pichler, and P. Guagliardo, Eds., ACM, pp. 325–341.

[36] Carmeli, N., Zeevi, S., Berkholz, C., Conte, A., Kimelfeld, B., and Schweikardt, N. Answering (unions of) conjunctive queries using random access and random-order enumeration. *ACM Trans. Database Syst. 47*, 3 (2022), 9:1–9:49.

[37] Chaudhuri, S., and Narasayya, V. R. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece* (1997), Morgan Kaufmann, pp. 146–155.

[38] Chen, B., Dai, B., Wang, Q., and Yi, K. Query running too slow? rewrite it with quorion! *Proc. VLDB Endow. 18*, 12 (2025), 5243–5246.

[39] Chen, H., and Dalmau, V. Decomposing quantified conjunctive (or disjunctive) formulas. In *Proceedings LICS* (2012), IEEE Computer Society, pp. 205–214.

[40] Chen, J., Huang, Y., Wang, M., Salihoglu, S., and Salem, K. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *SIGMOD Rec. 52*, 1 (2023), 94–102.

[41] Chen, J., Huang, Y., Wang, M., Salihoglu, S., and Salem, K. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *SIGMOD Rec. 52*, 1 (2023), 94–102.

[42] Colombo, P., and Ferrari, E. Efficient enforcement of action-aware purpose-based access control within relational database management systems. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016* (2016), IEEE Computer Society, pp. 1516–1517.

[43] Dai, B., Wang, Q., and Yi, K. Sparksql+: Next-generation query planning over spark. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023* (2023), S. Das, I. Pandis, K. S. Candan, and S. Amer-Yahia, Eds., ACM, pp. 115–118.

[44] Dalmau, V., and Jonsson, P. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci. 329*, 1-3 (2004), 315–323.

[45] Díaz, J., Serna, M. J., and Thilikos, D. M. Counting h-colorings of partial k-trees. *Theor. Comput. Sci. 281*, 1-2 (2002), 291–309.

[46] Dieu, N., Dragusanu, A., Fabret, F., Llirbat, F., and Simon, E. 1, 000 tables inside the from. *Proc. VLDB Endow. 2*, 2 (2009), 1450–1461.

[47] Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D. B., and Kraska, T. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* (2020), ACM, pp. 969–984.

[48] Dökeroglu, T., Bayir, M. A., and Cosar, A. Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries. *Appl. Soft Comput. 30* (2015), 72–82.

[49] Dong, Y., Indyk, P., Razenshteyn, I. P., and Wagner, T. Learning sublinear-time indexing for nearest neighbor search. *CoRR abs/1901.08544* (2019).

[50] Duggan, J., Çetintemel, U., Papaemmanouil, O., and Upfal, E. Performance prediction for concurrent database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011* (2011), T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, Eds., ACM, pp. 337–348.

[51] Durand, A., and Mengel, S. Structural tractability of counting of solutions to conjunctive queries. *Theory Comput. Syst. 57*, 4 (2015), 1202–1249.

[52] Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V. R., and Chaudhuri, S. Selectivity estimation for range predicates using lightweight models. *Proc. VLDB Endow. 12*, 9 (2019), 1044–1057.

[53] Feng, Y., You, H., Zhang, Z., Ji, R., and Gao, Y. Hypergraph neural networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019* (2019), AAAI Press, pp. 3558–3565.

[54] Fischl, W., Gottlob, G., Longo, D. M., and Pichler, R. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics 26* (2021), 1.6:1–1.6:40.

[55] Geck, G., Keppeler, J., Schwentick, T., and Spinrath, C. Rewriting with acyclic queries: Mind your head. In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)* (2022), D. Olteanu and N. Vortmeier, Eds., vol. 220 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:20.

[56] Gottlob, G., Lanzinger, M., Longo, D. M., Okulmus, C., Pichler, R., and Selzer, A. Reaching back to move forward: Using old ideas to achieve a new level of query optimization (short paper). In *Proceedings of the 15th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2023), Santiago de Chile, Chile, May 22-26, 2023* (2023), B. Kimelfeld, M. V. Martinez, and R. Angles, Eds., vol. 3409 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[57] Gottlob, G., Lanzinger, M., Longo, D. M., Okulmus, C., Pichler, R., and Selzer, A. Structure-guided query evaluation: Towards bridging the gap from theory to practice. *CoRR abs/2303.02723* (2023).

[58] Gottlob, G., Lanzinger, M., Okulmus, C., and Pichler, R. Fast parallel hypertree decompositions in logarithmic recursion depth. *ACM Trans. Database Syst. 49*, 1 (2024), 1:1–1:43.

[59] Gottlob, G., Lanzinger, M., Pichler, R., and Razgon, I. Fractional covers of hypergraphs with bounded multi-intersection. In *Proc. MFCS* (2020), vol. 170 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 41:1–41:14.

[60] Gottlob, G., Lanzinger, M., Pichler, R., and Razgon, I. Complexity analysis of generalized and fractional hypertree decompositions. *J. ACM 68*, 5 (2021), 38:1–38:50.

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

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

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

[64] Gottlob, G., Miklós, Z., and Schwentick, T. Generalized hypertree decompositions: Np-hardness and tractable variants. *J. ACM 56*, 6 (2009), 30:1–30:32.

[65] Gottlob, G., Okulmus, C., and Pichler, R. Fast and parallel decomposition of constraint satisfaction problems. *Constraints An Int. J. 27*, 3 (2022), 284–326.

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

[67] Gottlob, G., Scarcello, F., and Sideri, M. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artif. Intell. 138*, 1-2 (2002), 55–86.

[68] Graham, M. H. On The Universal Relation. Tech. rep., University of Toronto, 1979.

[69] GraphDB. https://graphdb.ontotext.com/, (accessed July 2024).

[70] Grasmann, L., Pichler, R., and Selzer, A. Integration of skyline queries into spark SQL. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023* (2023), J. Stoyanovich, J. Teubner, N. Mamoulis, E. Pitoura, J. Mühlig, K. Hose, S. S. Bhowmick, and M. Lissandrini, Eds., OpenProceedings.org, pp. 337–350.

[71] Green, T. J., Karvounarakis, G., and Tannen, V. Provenance semirings. In *Proceedings PODS* (2007), ACM, pp. 31–40.

[72] Grohe, M. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM 54*, 1 (2007), 1:1–1:24.

[73] Grohe, M., and Marx, D. Constraint solving via fractional edge covers. *ACM Trans. Algorithms 11*, 1 (2014), 4:1–4:20.

[74] Grohe, M., and Marx, D. Constraint solving via fractional edge covers. *ACM Trans. Algorithms 11*, 1 (2014), 4:1–4:20.

[75] Grushka-Cohen, H., Biller, O., Sofer, O., Rokach, L., and Shapira, B. Diversifying database activity monitoring with bandits. *CoRR abs/1910.10777* (2019).

[76] Han, Y., Wu, Z., Wu, P., Zhu, R., Yang, J., Tan, L. W., Zeng, K., Cong, G., Qin, Y., Pfadler, A., Qian, Z., Zhou, J., Li, J., and Cui, B. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *Proc. VLDB Endow. 15*, 4 (2021), 752–765.

[77] He, B., and Ounis, I. Query performance prediction. *Inf. Syst. 31*, 7 (2006), 585–594.

[78] Heimel, M., Kiefer, M., and Markl, V. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), ACM, pp. 1477–1492.

[79] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. Starfish: A self-tuning system for big data analytics. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings* (2011), www.cidrdb.org, pp. 261–272.

[80] Himmelstein, D. S., Lizee, A., Hessler, C., Brueggeman, L., Chen, S. L., Hadley, D., GreenP, A., ouya Khankhanian, and Baranzini, S. E. Systematic integration of biomedical knowledge prioritizes drugs for repurposing. *eLife 6*, e26726 (2017), 1–35.

[81] Hu, X., and Wang, Q. Computing the difference of conjunctive queries efficiently. *CoRR abs/2302.13140* (2023).

[82] Hu, X., and Wang, Q. Computing the difference of conjunctive queries efficiently. *Proc. ACM Manag. Data 1*, 2 (2023), 153:1–153:26.

[83] Hu, Z., and Miranker, D. P. Treetracker join: Turning the tide when a tuple fails to join. *CoRR abs/2403.01631* (2024).

[84] Idris, M., Ugarte, M., and Vansummeren, S. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017* (2017), S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds., ACM, pp. 1259–1274.

[85] Idris, M., Ugarte, M., Vansummeren, S., Voigt, H., and Lehner, W. General dynamic yannakakis: conjunctive queries with theta joins under updates. *VLDB J. 29*, 2-3 (2020), 619–653.

[86] Ji, X., Zhao, M., Zhai, M., and Wu, Q. Query execution optimization in spark SQL. *Sci. Program. 2020* (2020), 6364752:1–6364752:12.

[87] Jin, E., Bronstein, M., Ceylan, İ. İ., and Lanzinger, M. Homomorphism counts for graph neural networks: All about that basis. *CoRR abs/2402.08595* (2024).

[88] Jin, E., Bronstein, M. M., Ceylan, İ. İ., and Lanzinger, M. Homomorphism counts for graph neural networks: All about that basis. In *Proceedings ICML* (2024), OpenReview.net.

[89] Jin, G., Feng, X., Chen, Z., Liu, C., and Salihoglu, S. Kùzu graph database management system. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023* (2023), www.cidrdb.org.

[90] Joglekar, M. R., Puttagunta, R., and Ré, C. AJAR: aggregations and joins over annotated relations. In *Proceedings PODS* (2016), ACM, pp. 91–106.

[91] Khamis, M. A., Ngo, H. Q., and Rudra, A. FAQ: questions asked frequently. In *Proceedings PODS* (2016), ACM, pp. 13–28.

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

[93] Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P. A., and Kemper, A. Learned cardinalities: Estimating correlated joins with deep learning. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019), www.cidrdb.org.

[94] Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., and Shaikhha, A. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J. 23*, 2 (2014), 253–278.

[95] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. In *Proceedings of the 2018 International Conference*

on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018 (2018), ACM, pp. 489–504.

[96] Lamb, A., Shen, Y., Heres, D., Chakraborty, J., Kabak, M. O., Hsieh, L., and Sun, C. Apache arrow datafusion: A fast, embeddable, modular analytic query engine. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024* (2024), P. Barceló, N. Sánchez-Pi, A. Meliou, and S. Sudarshan, Eds., ACM, pp. 5–17.

[97] Lanzinger, M., Okulmus, C., Pichler, R., Selzer, A., and Gottlob, G. Soft and constrained hypertree width. *Proc. ACM Manag. Data 3*, 2 (2025), 114:1–114:25.

[98] Lanzinger, M., Pichler, R., and Selzer, A. Avoiding materialisation for guarded aggregate queries. In *Proceedings of the 16th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2024), Mexico City, Mexico, October 2-4, 2024* (2024), G. Montoya, E. Sallinger, and G. Vargas-Solar, Eds., vol. 3954 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[99] Lanzinger, M., Pichler, R., and Selzer, A. Avoiding materialisation for guarded aggregate queries. *CoRR abs/2406.17076* (2024).

[100] Lanzinger, M., Pichler, R., and Selzer, A. Avoiding materialisation for guarded aggregate queries. *Proc. VLDB Endow. 18*, 5 (2025), 1398–1411.

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

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

[103] Leskovec, J., and Sosic, R. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol. 8*, 1 (2016), 1:1–1:20.

[104] Li, G., Zhou, X., Li, S., and Gao, B. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow. 12*, 12 (2019), 2118–2130.

[105] Lodeiro-Santiago, M., Caballero-Gil, C., and Caballero-Gil, P. Collaborative sql-injections detection system with machine learning. In *Proceedings of the 1st International Conference on Internet of Things and Machine Learning, IML 2017, Liverpool, United Kingdom, October 17-18, 2017* (2017), H. Hamdan, D. E. Boubiche, and F. Klett, Eds., ACM, pp. 45:1–45:5.

[106] Lutz, C., and Przybylko, M. Efficiently enumerating answers to ontology-mediated queries. In *PODS '22: International Conference on Management of Data,*

Philadelphia, PA, USA, June 12 - 17, 2022 (2022), L. Libkin and P. Barceló, Eds., ACM, pp. 277–289.

[107] Ma, L., Aken, D. V., Hefny, A., Mezerhane, G., Pavlo, A., and Gordon, G. J. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), ACM, pp. 631–645.

[108] Ma, M., Yin, Z., Zhang, S., Wang, S., Zheng, C., Jiang, X., Hu, H., Luo, C., Li, Y., Qiu, N., Li, F., Chen, C., and Pei, D. Diagnosing root causes of intermittent slow queries in large-scale cloud databases. *Proc. VLDB Endow. 13*, 8 (2020), 1176–1189.

[109] Mageirakos, V., Mancini, R., Karthik, S., Chandra, B., and Ailamaki, A. Efficient gpu-accelerated join optimization for complex queries. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022* (2022), IEEE, pp. 3190–3193.

[110] Mancini, R., Karthik, S., Chandra, B., Mageirakos, V., and Ailamaki, A. Efficient Massively Parallel Join Optimization for Large Queries. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data, SIGMOD Conference 2022* (2022), ACM, pp. 122–135.

[111] Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., and Kraska, T. Bao: Making learned query optimization practical. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, pp. 1275–1288.

[112] Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., and Kraska, T. Bao: Making learned query optimization practical. *SIGMOD Rec. 51*, 1 (2022), 6–13.

[113] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., and Tatbul, N. Neo: A learned query optimizer. *Proc. VLDB Endow. 12*, 11 (2019), 1705–1718.

[114] Marcus, R., and Papaemmanouil, O. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow. 12*, 11 (2019), 1733–1746.

[115] Marx, D. Can you beat treewidth? *Theory Comput. 6*, 1 (2010), 85–112.

[116] Mhedhbi, A., Lissandrini, M., Kuiper, L., Waudby, J., and Szárnyas, G. LSQB: a large-scale subgraph query benchmark. In *Proceedings GRADES* (2021), ACM, pp. 8:1–8:11.

[117] Mhedhbi, A., Lissandrini, M., Kuiper, L., Waudby, J., and Szárnyas, G. LSQB: a large-scale subgraph query benchmark. In *Proc. GRADES* (2021), ACM, pp. 8:1–8:11.

[118] Misegiannis, M. G., Kantere, V., and d'Orazio, L. Multi-objective query optimization in spark SQL. In *Proceedings IDEAS* (2022), ACM, pp. 70–74.

[119] Mizzaro, S., Mothe, J., Roitero, K., and Ullah, M. Z. Query performance prediction and effectiveness evaluation without relevance judgments: Two sides of the same coin. In *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, SIGIR 2018, Ann Arbor, MI, USA, July 08-12, 2018* (2018), K. Collins-Thompson, Q. Mei, B. D. Davison, Y. Liu, and E. Yilmaz, Eds., ACM, pp. 1233–1236.

[120] Neo4J. https://neo4j.com/, (accessed July 2024).

[121] Neumann, T., and Freitag, M. J. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings* (2020), www.cidrdb.org.

[122] Neumann, T., and Radke, B. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 677–692.

[123] Nguyen, H., and Maehara, T. Graph homomorphism convolution. In *Proceedings ICML* (2020), vol. 119 of *Proceedings of Machine Learning Research*, PMLR, pp. 7306–7316.

[124] Nguyen, N., Khan, M. M. H., and Wang, K. Towards automatic tuning of apache spark configuration. In *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018* (2018), IEEE Computer Society, pp. 417–425.

[125] Nikolic, M., and Olteanu, D. Incremental view maintenance with triple lock factorization benefits. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 365–380.

[126] Olteanu, D., and Schleich, M. Factorized databases. *SIGMOD Rec. 45*, 2 (2016), 5–16.

[127] Olteanu, D., and Závodný, J. Size bounds for factorised representations of query results. *ACM Trans. Database Syst. 40*, 1 (2015), 2:1–2:44.

[128] Ortiz, J., Balazinska, M., Gehrke, J., and Keerthi, S. S. An empirical analysis of deep learning for cardinality estimation. *CoRR abs/1905.06425* (2019).

[129] Park, Y., Zhong, S., and Mozafari, B. Quicksel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* (2020), ACM, pp. 1017–1033.

[130] Perelman, A., and Ré, C. Duncecap: Compiling worst-case optimal query plans. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015* (2015), T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds., ACM, pp. 2075–2076.

[131] Pichler, D. B. G. G. M. L. D. L. C. O. R., and Selzer, A. Selective use of yannakakis' algorithm to improve query performance: Machine learning to the rescue. *CoRR abs/2502.20233* (2025).

[132] Pichler, R., and Skritek, S. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci. 79*, 6 (2013), 984–1001.

[133] Pöss, M., Smith, B., Kollár, L., and Larson, P. Tpc-ds, taking decision support benchmarking to the next level. In *Proc. SIGMOD* (2002), ACM, pp. 582–587.

[134] Raasveldt, M., and Mühleisen, H. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019* (2019), ACM, pp. 1981–1984.

[135] Ravid, N., Medini, D., and Kimelfeld, B. Ranked enumeration of minimal triangulations. In *Proc. PODS* (2019), ACM, pp. 74–88.

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

[137] Schidler, A., and Szeider, S. Computing optimal hypertree decompositions. In *Proc. ALENEX* (2020), SIAM, pp. 1–11.

[138] Schidler, A., and Szeider, S. Computing optimal hypertree decompositions with SAT. *Artif. Intell. 325* (2023), 104015.

[139] Schleich, M., Olteanu, D., Khamis, M. A., Ngo, H. Q., and Nguyen, X. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds., ACM, pp. 1642–1659.

[140] Schnaitter, K., Abiteboul, S., Milo, T., and Polyzotis, N. On-line index selection for shifting workloads. In *Proceedings of the 23rd International Conference on Data Engineering Workshops, ICDE 2007, 15-20 April 2007, Istanbul, Turkey* (2007), IEEE Computer Society, pp. 459–468.

[141] Seshadri, P., Pirahesh, H., and Leung, T. Y. C. Complex query decorrelation. In *Proc. ICDE'96* (1996), IEEE Computer Society, pp. 450–458.

[142] Shaikhha, A., Huot, M., Smith, J., and Olteanu, D. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang. 6*, OOPSLA1 (2022), 1–33.

[143] Shen, Y., Ren, X., Lu, Y., Jiang, H., Xu, H., Peng, D., Li, Y., Zhang, W., and Cui, B. Rover: An online spark SQL tuning service via generalized transfer learning. In *Proceedings KDD* (2023), ACM, pp. 4800–4812.

[144] Sheykhkanloo, N. M. A learning-based neural network model for the detection and classification of SQL injection attacks. *Int. J. Cyber Warf. Terror. 7*, 2 (2017), 16–41.

[145] Stillger, M., Lohman, G. M., Markl, V., and Kandil, M. LEO - db2's learning optimizer. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy* (2001), Morgan Kaufmann, pp. 19–28.

[146] Stonebraker, M., and Kemnitz, G. The postgres next generation database management system. *Commun. ACM 34*, 10 (1991), 78–92.

[147] Taft, R., El-Sayed, N., Serafini, M., Lu, Y., Aboulnaga, A., Stonebraker, M., Mayerhofer, R., and Andrade, F. J. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), G. Das, C. M. Jermaine, and P. A. Bernstein, Eds., ACM, pp. 205–219.

[148] Tan, J., Zhang, T., Li, F., Chen, J., Zheng, Q., Zhang, P., Qiao, H., Shi, Y., Cao, W., and Zhang, R. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proc. VLDB Endow. 12*, 10 (2019), 1221–1234.

[149] TPC-DS. Tpc-ds benchmark. `https://www.tpc.org/tpcds/`.

[150] TPC-H. Tpc-h benchmark. `https://www.tpc.org/tpch/`.

[151] Trummer, I., Wang, J., Wei, Z., Maram, D., Moseley, S., Jo, S., Antonakakis, J., and Rayabhari, A. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. *ACM Trans. Database Syst. 46*, 3 (2021), 9:1–9:45.

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

[153] Tzoumas, K., Sellis, T., and Jensen, C. S. A reinforcement learning approach for adaptive query processing. *History* (2008), 1–25.

[154] Wang, Q., Chen, B., Dai, B., Yi, K., Li, F., and Lin, L. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proc. ACM Manag. Data 3*, 3 (2025), 235:1–235:28.

[155] Wang, Q., Hu, X., Dai, B., and Yi, K. Change propagation without joins. *CoRR abs/2301.04003* (2023).

[156] Wang, Q., and Yi, K. Conjunctive queries with comparisons. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022* (2022), Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds., ACM, pp. 108–121.

[157] Wang, Y., and Yi, K. Secure yannakakis: Join-aggregate queries over private data. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, pp. 1969–1981.

[158] Wang, Z., Zhou, Z., Yang, Y., Ding, H., Hu, G., Ding, D., Tang, C., Chen, H., and Li, J. Wetune: Automatic discovery and verification of query rewrite rules. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022* (2022), Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds., ACM, pp. 94–107.

[159] Wu, W., Chi, Y., Zhu, S., Tatemura, J., Hacigümüs, H., and Naughton, J. F. Predicting query execution time: Are optimizer cost models really unusable? In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013* (2013), C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds., IEEE Computer Society, pp. 1081–1092.

[160] Wu, Y., Yu, J., Tian, Y., Sidle, R., and Barber, R. Designing succinct secondary indexing mechanism by exploiting column correlations. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), ACM, pp. 1223–1240.

[161] Yang, Y., Zhao, H., Yu, X., and Koutris, P. Predicate transfer: Efficient pre-filtering on multi-join queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024* (2024), www.cidrdb.org.

[162] Yannakakis, M. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Databases, VLDB 1981, Cannes* (1981), VLDB, pp. 82–94.

[163] Yu, C. T., and zsoyoglu, M. Z. O. An algorithm for tree-query membership of a distributed query. In *The IEEE Computer Society's Third International Computer Software and Applications Conference, COMPSAC 1979* (1979), pp. 306–312.

[164] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache spark: a unified engine for big data processing. *Commun. ACM 59*, 11 (2016), 56–65.

[165] Zhai, M., Song, A., Qiu, J., Ji, X., and Wu, Q. Query optimization approach with shuffle intermediate cache layer for spark SQL. In *Proceedings IPCCC* (2019), IEEE, pp. 1–6.

[166] Zhang, H., Yu, J. X., Zhang, Y., Zhao, K., and Cheng, H. Distributed subgraph counting: A general approach. *Proc. VLDB Endow. 13*, 11 (2020), 2493–2507.

[167] Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., Ran, M., and Li, Z. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019* (2019), ACM, pp. 415–432.

[168] Zhang, N., Haas, P. J., Josifovski, V., Lohman, G. M., and Zhang, C. Statistical learning techniques for costing XML queries. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005* (2005), K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, Eds., ACM, pp. 289–300.

[169] Zhao, J., Su, K., Yang, Y., Yu, X., Koutris, P., and Zhang, H. Debunking the myth of join ordering: Toward robust SQL analytics. *Proc. ACM Manag. Data 3*, 3 (2025), 146:1–146:28.

[170] Zhou, X., Chai, C., Li, G., and Sun, J. Database meets artificial intelligence: A survey. *IEEE Trans. Knowl. Data Eng. 34*, 3 (2022), 1096–1116.

[171] Zhou, X., Li, G., Chai, C., and Feng, J. A learned query rewrite system using monte carlo tree search. *Proc. VLDB Endow. 15*, 1 (2021), 46–58.

[172] Zhou, X., Li, G., Wu, J., Liu, J., Sun, Z., and Zhang, X. A learned query rewrite system. *Proc. VLDB Endow. 16*, 12 (2023), 4110–4113.

[173] Zhou, X., Sun, J., Li, G., and Feng, J. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow. 13*, 9 (2020), 1416–1428.

[174] Zhou, Y., and Croft, W. B. Query performance prediction in web search environments. In *SIGIR 2007: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Amsterdam, The Netherlands, July 23-27, 2007* (2007), W. Kraaij, A. P. de Vries, C. L. A. Clarke, N. Fuhr, and N. Kando, Eds., ACM, pp. 543–550.

[175] Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., and Yang, Y. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017* (2017), ACM, pp. 338–350.