TU WIEN Informatics

Erzeugen von Guardedness für azyklische Aggregatabfragen zur Minimierung der Materialisierung von Zwischenergebnissen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andrea Ortner, BSc.
Matrikelnummer 11809650

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 22. April 2025

Andrea Ortner	Reinhard Pichler

TU WIEN Informatics

Enforcing Guardedness for Acyclic Aggregate Queries to Minimize Materialization

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

Andrea Ortner, BSc.
Registration Number 11809650

to the Faculty of Informatics

at the TU Wien

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

Vienna, April 22, 2025

Andrea Ortner | Reinhard Pichler

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

Erklärung zur Verfassung der Arbeit

Andrea Ortner, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
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, 22. April 2025

Andrea Ortner

Danksagung

Zuallererst möchte ich an dieser Stelle meinen Eltern für ihre stete Unterstützung in all den Jahren danken. Mama, Papa - es ist wahrlich ein Privileg, auf euren Schultern zu stehen. Danke, dass ihr meine Eskapaden und verrückten Ideen nicht nur ertragen, sondern sogar unterstützt habt.

Ein großes Dankeschön gebührt meinem Bruder und bestem Freund Chris. Mit dir in meiner Ecke steig ich in jeden Boxring.

Eine Bärenumarmung verdienen auch meine Freunde, mit denen ich sowohl die schönsten, als auch die schwierigsten Momente der letzten Studienjahre geteilt habe: Sonja, Fabian, Robert, Tahel.

Dankbar bin ich ebenfalls meinem Betreuer Prof. Dr. Reinhard Pichler und seinem Kollegen Dipl.-Ing. Alexander Selzer. Vom Themenvorschlag bis zum Abschluss der Arbeit konnte ich mich auf ihr "Coaching" verlassen und auf ihre Expertise und Hilfe zählen. Danke auch für die Schaffung von Arbeitsumständen, die es mir erlaubt haben, mich voll und ganz auf meine Diplomarbeit zu fokussieren.

Außerdem möchte ich mir selber danken, dafür, dass ich nie aufgegeben habe und die Challenges der letzten Jahre stets auch als Chance betrachtet habe.

Zum Schluss möchte ich kurz auch jenen Leuten ein paar Worte widmen, die meine Bestrebungen belächelt haben und die mir den Mut nehmen wollten. Wisset, dass eure Energien verschwendet waren. Vergessen werde ich euch nicht.

Acknowledgements

First and foremost, I want to thank my parents for their unwavering support. Mama, Papa - what a privilege it is to stand on your shoulders. Thank you for putting up with my antics and supporting my crazy little (and big) ideas.

A huge thank you also goes to my brother and best friend, Chris. With you in my corner, there is nothing to fear.

A warm hug goes out to my friends, with whom I have shared the best and worst academic moments over the past couple of years: Sonja, Fabian, Robert and Tahel.

I also want to thank my supervisor, Prof. Dr. Reinhard Pichler, and his colleague, Dipl.-Ing. Alexander Selzer, for suggesting the thesis topic, "coaching" me through the whole process and providing good working conditions that allowed me to fully focus on my thesis. I am grateful for your expertise and helpful suggestions whenever I thought I had hit a wall.

Furthermore, I want to thank myself for never giving up and for seeing the challenges of the last couple of years as opportunities to grow.

And lastly, I want to acknowledge all the people who belittled my endeavors and tried to take my courage away. Rest assured, your energy was wasted. You will not be forgotten.

Kurzfassung

Petabytes an Daten - das ist der Preis unserer globalisierten, digitalen Welt. Nur Hardware alleine kann diesen Berg an Daten nicht bewältigen, zumal ihre Mächtigkeit auch physisch limitiert ist (Moore's Law). Will man bestimmte Informationen aus diesen Daten abfragen, ist dies mit hohen Kosten verbunden. Um diese zu senken, ist es notwendig, intelligente Software-Lösungen zu entwickeln. Doch warum ist diese Beantwortung bzw. Evaluierung so kostspielig?

Ein Grund für die hohen anfallenden Kosten ist das Entstehen sehr großer Zwischenergebnisse bei relativ kleinen Endergebnissen. Dieses Problem konnte Yannakakis bereits 1981 etwas abschwächen, als er einen Algorithmus entwarf, der im Fall von Join-Abfragen nur jene Zwischenergebnisse materialisierte, die auch Teil des Endergebnisses waren. Im Fall von Aggregat-Abfragen bedeutet dies natürlich immer noch, dass große Zwischenergebnisse materialisiert werden, obwohl das Endergebnis nur aus den Aggregat(en) besteht. Die neueste Entwicklung bzgl. Yannakakis-artigen Optimierungen, genannt AggJoin-Operator [LPS24], vermeidet Materialisierung im Fall von azyklischen Aggregat-Abfragen komplett. AggJoin setzt dafür aber voraus, dass die zu beantwortenden Abfragen "piecewise-guarded" sind. Das bedeutet, dass alle Attribute in der GROUP BY Klausel bzw innerhalb eines Aggregat-Ausdrucks gemeinsam in einer der an den Joins beteiligten Relation vorkommen müssen. Dies stellt eine große Einschränkung dar, schließlich können so insbesondere Abfragen, deren GROUP BY Klauseln Attribute aus mehreren Relationen enthalten (ein häufiger Use-Case in Business Analytics), nicht mit AggJoin evaluiert werden. Aus diesem Grund ist es das Ziel dieser Arbeit, eine Brücke zwischen Yannakakis-artigen Optimierungen (AggJoin), und non-guarded Abfragen zu bauen (d.h. Abfragen, deren gruppierende Attribute aus verschiedenen Relationen stammen). Dazu haben wir PartAggJoin implementiert; eine Erweiterung von AggJoin, die nicht an dieselben Restriktionen gebunden ist: PartAggJoin zerlegt Abfragen-Bäume in Sub-Bäume, welche mit unterschiedlichen Methoden evaluiert werden. Sind gruppierende Attribute vorhanden, wird minimale Materialisierung erlaubt, um keine Informationen zu verlieren; im anderen Fall wird AggJoin angewandt. Dadurch können nun auch Abfragen mit mehreren Attributen im GROUP BY Teil von Yannakakis-artigen Optimierungsstrategien profitieren. Die Effizienz und Performanz unserer Optimierung wurde durch die Implementierung in SparkSQL bewiesen und empirisch anhand der Benchmarks TPC-H und Syn-TPC-H gezeigt. Syn-TPC-H wurde als Datenset im Rahmen dieser Arbeit geschaffen, um einen gezielten Fokus auf non-guarded Abfragen zu erlauben.

Abstract

Petabytes of data - such is the cost of our modern, global, digital interconnectedness. The approach of relying solely on hardware to process this amount of data is slowly reaching its limit (Moore's Law). This is why we need intelligent software solutions that reduce the energy, time and overall cost of data processing. But why is it even so costly?

One reason for the high cost of processing is the explosion of intermediate results, even if the final result is rather small. To mitigate this problem to some extent, Yannakakis designed an algorithm in 1981 that, when multiple joins are necessary, only materializes tuples that are actually part of the final result. In case of aggregate queries, this still means that intermediate results are quite big in comparison to the final result (the aggregated values). The state-of-the-art optimization (the AggJoin operator [LPS24]), provides powerful speed-gains and succeeded in avoiding materialization altogether for "piecewise-guarded" acyclic aggregate queries. This means, that all attributes in the GROUP BY statement need to stem from the same relation, and so do attributes within an aggregate expression. This is very limiting, as queries that have grouping attributes from multiple relations are a common use case in business analytics. For this reason, this work sought to build a bridge between Yannakakis-style query optimization and non-guarded queries by allowing minimal necessary materialization. To this end, we implemented PartAggJoin, an extension to AggJoin that lifts the restrictions in terms of guardedness: PartAggJoin splits a query tree into subtrees. During the subsequent analysis of these subtrees, PartAggJoin detects whether a subtree contains applicable grouping attributes. If this is the case, it performs the minimal materialization necessary to preserve the grouping attributes. If no grouping attributes are present, an AggJoin is performed. Thus, PartAggJoin removes the limitations of guardedness for queries that contain two or more attributes in GROUP BY clauses that stem from different relations, widening the field of application for Yannakakis-style optimization. The efficiency and performance power of our optimization is validated through its implementation in SparkSQL and empirical evaluation on the TPC-H benchmark as well as on the newly created Syn-TPC-H dataset that specializes on the evaluation of non-guarded queries.

Contents

CHAPTER 1

Introduction

Background. Big Data sounds great until we actually come face to face with the problems of *processing* said big data. Even though modern Database Management Systems (DBMS) are quite powerful, they are not able to handle queries with joins regarding up to hundreds of relations (in fact, even handling 10 joins is already a difficult task). This is because the two most prominent problems in query optimization, namely finding a good join order, which is NP-complete, and avoiding the explosion of intermediate results, can only be tackled by relying on heuristics, at least after a certain threshold of involved relations is reached. And the greater the amount of involved relations, the lesser the quality of the proposed optimization [GLL+23a]. The problem of exploding intermediate results is especially bothersome in fields like business analytics, that often feature aggregate queries that involve a great amount of joins (and thus huge intermediate results), but results are typically only a few aggregated values.

In 1981, Yannakakis proposed an algorithm to reduce the size of intermediate results for a certain subclass of queries, namely acyclic conjunctive queries (ACQs) (see Chapter 2 and Chapter 3). By relying on semi-joins and eliminating tuples which are not part of the final results, this method materializes only necessary tuples [Yan81]. This process allows evaluating acyclic conjunctive queries (ACQs) within three traversals of a join tree in linear time w.r.t the input, output and query size.

For boolean queries, it was shown that the algorithm can be stopped after the first bottom-up traversal [GLS01]. In subsequent works, a new class of queries, called zero-materialization-queries (0MA) have been identified, for which it is also sufficient to do one bottom-up traversal of semi-joins [GLL+23a], without materializing any join. However, these 0MA queries have to satisfy three strict conditions (additionally to being ACQ): The query has to be in aggregation normal form (i.e. apart from aggregation, grouping and projection a query consists only of natural joins and selections), guarded (at least one relation needs to hold all grouping and aggregation attributes) and set-safe [GLL+23a]. Set-safety is especially restrictive, since aggregations like SUM, AVG, MEDIAN and

COUNT(*) are per definition excluded (see Chapter 4).

Problem Statement. The above-mentioned problem regarding set-safety has been mitigated by more recent work [LPS24]. By introducing a new physical operator - the AggJoin - that allows for propagating frequencies of tuples instead of materializing intermediate results, Lanzinger et al. have widened the field of application for Yannakakis-style query optimization even more. However, queries still have to have two quite restrictive properties in order to profit from the newest optimizations: They have to be acyclic and guarded or at least piecewise-guarded (see Section 4.3). [LPS24].

Since it has been shown empirically [BMT20, FGLP21] that in practice one mostly encounters acyclic or almost acyclic queries, we have decided to take on the problem of enforcing guardedness for ACQs. Three methods have been identified to achieve this: The *Upfront Joins* method, the *Partial AggJoin (PartAggJoin)* and another method called *GroupAggJoin*. Whereas the *Upfront Joins* method and *PartAggJoin* focus less on *avoiding* full materialization, but rather tries to *minimize* materialization in order to make the optimization proposed in [LPS24] applicable to non-guarded ACQs, *GroupAggJoin* works similar to AggJoin, but additionally to propagating aggregates (see Section 4.5), it also propagates attributes in the GROUP BY statement. We will describe these three approaches in Section 6.2. Since *GroupAggJoin* is currently developed by the authors of [LPS24], we will compare the other two approaches and implement the method that proves more suitable in a SparkSQL environment (in our case, this is *PartAggJoin*, see Section 6.2). Eventually we will provide benchmarks of the actual implementation of *PartAggJoin* in a Spark environment in Section 6.4.

An improvement to existing systems is needed, because even though the optimization proposed in [LPS24] has shown to lead to great speed-gains, it is only applicable to a limited subset of queries, as was seen in the benchmark of the same paper: AggJoin was applicable to all queries in the STATS-CEB dataset [HWW+21] and the SNAP dataset [LK14]. Also, all 113 JOB [LGM+15] queries could be processed by the new optimization. However, only 7 out of 22 TPC-H [TPCb] queries, 30 out of 99 TPC-DS [TPCa] queries and 2 out of 9 LSQB [MLK+21] queries could be processed with AggJoin. Since big data has shown again and again that it cannot be handled by (only) strong hardware, it is paramount to further improve current optimizations and widen their field of application in order to save space, time and costs and to ensure operability and reliability of future DBMS.

Goal of the thesis. The over-arching goal of this thesis is to widen the field of application for AggJoin, the query optimization proposed by Lanzinger et al. [LPS24]. For this, the three methods of extending the current algorithm mentioned above will be described and compared. As stated above, the first two methods, *Upfront Joins* and *PartAggJoin*, will allow minimal materialization in order to transform non-guarded queries into guarded ones, which makes them eligible for the optimizations proposed in [LPS24]. In case of *Upfront Joins*, such minimal materialization occurs in joins that

are executed before the query gets processed in order to create an auxiliary table that acts like a guard for the query (hence the name *Upfront Joins*). The second method, *PartAggJoin*, will also use full materialization, but only in places where it is necessary to propagate attributes that appear in the GROUP BY statement. The third method will extend the current AggJoin implementation by propagating not only aggregate columns, but also columns that appear in the GROUP BY statement, creating the GroupAggJoin operator. Regarding the practical implementation, GroupAggJoin is being implemented by Alexander Selzer [LPS25], whereas the methods of *Upfront Joins* and *PartAggJoin* will be examined as part of this work.

In particular, the following research questions have been investigated in our contribution:

1. How can we algorithmically restructure non-guarded ACQs into guarded ACQs in a both time and space efficient way in order to make them eligible for the query optimization proposed in [LPS24]?

2. How can we extend the state-of-the-art implementation of AggJoin presented in [LPS24] for query optimization to make it more applicable to real-world scenarios?

The goal is to find a generally-applicable algorithmic solution that transforms non-guarded queries into (piecewise-)guarded ones. Upon finding the most suitable algorithm, we will implement it as an extension for the AggJoin-Procedure presented in [LPS24]. Eventually, we will add our own benchmark dataset that contains queries that are non-guarded, but will be transformed into guarded queries by our implementation. Since our contribution allows optimization of a bigger share of the benchmark queries, we expect that a) more queries profit from the optimization proposed in [LPS24] evaluation and b) the overall speed-gain of the benchmark sets improves.

Approach. The research plan of this thesis contains the following steps: analyzing the benchmark data and extending it with new queries, finding an appropriate algorithm to enforce guardedness by comparing different approaches (*Upfront Joins*, *PartAggJoin* and *GroupAggJoin* [LPS25]) and finally the evaluation of our implementation. As preparatory first step, appropriate literature will be selected. Topics of interest in this step will be traditional approaches to join-evaluation and query processing, as well as more specific research, such as structure guided query evaluation, 0MA queries and guarded aggregate queries. Additionally, the theoretical groundwork needed to extend the given implementation will be laid by researching the aspects of CQs, hypertrees and various query structures.

In the next step, we will analyze the query structures in the current benchmark data, that is JOB [LGM+15], STATS [HWW+21]SNAP [LK14], TPC-H [TPCb], TPC-DS [TPCa] and LSQB [MLK+21], i.e. we will have a look on how many relations are joined on average, how many grouping attributes appear and so on in order to decide whether it is also sensible to use the same data for our benchmarks. Also, we will create a new

synthetic dataset based on the datasets of TPC-H [TPCb] called Syn-TPC-H in order to specifically target non-guarded queries.

Step 3 of the research consist of comparing the three variants, namely *Upfront Joins*, *PartAggJoin* and *GroupAggjoin*, and finally extending the current work of [LPS24]. The authors have implemented their optimization in SparkSQL by extending SparkSQL's source code written in Scala. Without going into too much detail, we will quickly sketch where our work will be included in their program: All queries are transformed into a join-tree. Whenever a join treeT or subtree T' of a join treeis given, the first step of the algorithm is to re-structure T (or T') so that the root node acts as a guard to the tree, i.e. it holds all attributes that are needed by the GROUP BY operator. Currently, if no such guard exists, the optimization is not applied. Our goal is to implement an extension to the current algorithm that creates such a guard (more details can be found in Section 6.2 and Section 6.3. To the best of our knowledge, no single, widely established algorithm that universally converts a non-guarded query to a guarded one exists.

In step 4 of the thesis, we will evaluate the chosen extension on the TPC-H benchmark that was also used in [LPS24], as well as our synthetic dataset Syn-TPC-H.

Results. In this thesis, we will present a method to apply Yannakakis-style optimization to non-guarded queries. After comparing the methods mentioned above, we found that *PartAggJoin* currently the best suited approach for a Spark environment for various reasons (see Section 6.2).

Our implementation of *PartAggJoin* is able to evaluate all non-guarded queries of the new Syn-TPC-H dataset and more than half of the non-guarded queries in the TPC-H dataset. It is able to process non-guarded queries using Yannakakis-style optimization by splitting the join treeinto subtrees. Based on whether these subtrees contain relations that have grouping attributes, the algorithm either uses AggJoin or chooses to allow minimal materialization. Since *PartAggJoin* is an extension for AggJoin, it is applicable for the same aggregate functions as AggJoin, namely MIN, MAX, SUM, COUNT and AVG.

Structure of the thesis. We will begin our work by giving some theoretical background. In Chapter 2 we will talk about CQs, their characteristics and the complexity of evaluating such queries. In Chapter 3, we will formally introduce hypergraphs and explain their links to acyclicity and join trees. We will also touch upon the GYO-reduction used to detect acyclicity and explain Yannakakis' algorithm [Yan81] in detail. Chapter 4 will focus on newer developments regarding Yannakakis-style query optimization, namely complex queries like 0MA queries [GLL+23a] and guarded aggregate queries [LPS24]. Afterwards, we give a short introduction to Spark and SparkSQL in Chapter 5. In Chapter 6 the methodology, including data analysis, finding a suitable algorithmic solution and implementation thereof are described, followed by Chapter 7 where we present and discuss the obtained results. Finally the results, limitations and future work are summarized in Chapter 8.

CHAPTER 2

Conjunctive Queries

In this chapter, we will introduce a basic type of queries, namely Conjunctive Queries (CQs), that every DBMS has to be able process. In Section 2.1 we will talk about common structures, properties and different notation styles of CQs. Section 2.2 then focuses on the computational complexity of evaluating CQs.

As basis for the given examples we define the database schema R, containing the relations *DOG*, *OWNER*, *MEDICATION* and *VISIT*, so R = {DOG, OWNER, MEDICATION, VISIT}. The attributes of these relations can be seen in Table 2.1, Table 2.2, Table 2.3 and Table 2.4. In the following section we will abbreviate the relations with their first letters, i.e. DOG = D, OWNER = 0, MEDICATION = M, VISIT = V.

Table 2.1: Attributes of the relation DOG.

D_ID	Name	Breed	Weight	Medication	O_ID

Table 2.2: Attributes of the relation OWNER.

O_ID	Firstname	Lastname	Address	Bank	Telephone

Table 2.3: Attributes of the relation MEDICATION.

M_ID	Name	Brand	Dosage	Price

Table 2.4: Attributes of the relation VISIT.

V_ID	O_ID	D_ID	Date	Payment

2.1 Conjunctive Queries (CQs)

Conjunctive queries (CQs) are a fundamental class of query types. In Relational Algebra (RA), they typically correspond to select-project-join queries. [CM77].

Formal structure:

$$\pi(\sigma(R_1 \bowtie ... \bowtie R_i))$$

Example:

$$\pi(\sigma(D \bowtie O))$$

Importantly, the applied join is always an equi-join or natural join. Furthermore, the query can only contain equality and conjunction as operators. In the next two sections, we will list important properties of CQs and show different notation styles that go beyond RA.

2.1.1 Properties of CQs

Rewriting without Selections. CQs can be rewritten without explicit selection operations (σ) if only equality conditions are present. If the equality condition includes two different relations, the selection is treated as an equi-join. The selection criteria can be added to the join as an additional join condition, thus eliminating the need for a separate selection.

Formal structure:

$$\pi(\sigma_{R_1.x=R_2.y}(R_1 \bowtie_{R_1.a=R_2.b} R_2 ... R_{i-1} \bowtie_{R_{i-1}.n=R_i.m} R_i)) \Leftrightarrow \pi(R_1 \bowtie_{R_1.a=R_2.b \wedge R_1.x=R_2.y} R_2 ... R_{i-1} \bowtie_{R_{i-1}.n=R_i.m} R_i)$$

Example:

$$\pi(\sigma_{D.o\_id=V.o\_id}(D \bowtie_{D.d\_id=V.d\_id} V)) \Leftrightarrow \pi(D \bowtie_{D.o\_id=V.o\_id \wedge D.d\_id=V.d\_id} V)$$

If the selection only involves the column(s) of one relation, e.g. $\sigma_{R_1.x=5}$, the selection can also be omitted by executing the selection before the join as a separate filtering step. Assuming $\tilde{R_1} = \sigma_{R_1.x=5}(R_1)$, the following transformation is possible:

Formal structure:

$$\pi(\sigma_{R_1.x=5}(R_1 \bowtie_{R_1.a=R_2.b} R_2) ... R_{i-1} \bowtie_{R_{i-1}.n=R_i.m} R_i) \Leftrightarrow \pi(\tilde{R_1} \bowtie_{R_1.a=R_2.b} R_2 ... R_{i-1} \bowtie_{R_{i-1}.n=R_i.m} R_i)$$

Example:

$$\pi(\sigma_{D.breed=\text{"}Westi\text{"}}(D \bowtie_{D.o\_id=O.o\_id} O)) \Leftrightarrow \pi(\tilde{D} \bowtie_{D.o\_id=O.o\_id} O)$$

Equivalence of Equi-Join and Natural Joins. Equi-joins with equality conditions can be represented as natural joins after renaming columns appropriately. This can be demonstrated with an example where a column is renamed (ρ) to make the equi-join equivalent to a natural join.

Formal structure:

$$\pi(R_1 \bowtie_{R_1.a=R_2.b} R_2) \Leftrightarrow \pi((\rho_{b \leftarrow a} R_1) \bowtie R_2)$$

Example:

$$\pi(D \bowtie_{D.ownerid=O.id} O) \Leftrightarrow \pi(D) \bowtie (\rho_{ownerid \leftarrow id} O)$$

2.1.2 CQs in Different Notation Styles

As already written above, CQs in RA are formally written as

$$\pi(\sigma(R_1 \bowtie R_2 ... \bowtie R_i))$$

Since most DBMS are based on some SQL dialect, we will also look at CQs written in SQL. The example $\pi_{R_1.c,R_2.d}(R_1 \bowtie_{R_1.a=R_2.b} R_2)$ corresponds to the following SQL query:

```
SELECT R1.c, R2.d
FROM R1, R2
WHERE R1.a = R2.b
```

As concrete example in our database schema, the query $\pi_{D.name,O.lastname}(D \bowtie_{D.ownerid=O.id} O)$ corresponds to the following SQL query:

```
SELECT D.name, O.lastname
FROM DOG as D, OWNER as O
WHERE D.o_id = O.o_id
```

Lastly, one example of a CQ in datalog notation is given. For this example, we add some schema information to the example from above. Assume that the relation R_1 has two attributes, namely a and c, so it can be written as $R_1[a,c]$. Correspondingly, R_2 has two columns, namely b and d and can thus be written as $R_2[b,d]$. Every datalog query consists of the head, where the projection occurs, and the body containing atoms (the relations) that correspond to joins and filtering. The translation of the SQL query from above into datalog yields the following result:

$$Q(c, d) : - R_1(x, c), R_2(x, d)$$

Since we know the schema structure of the relations, we can translate the query by selecting every tuple ($R_1.c$, $R_2.d$) where the *first attributes* of the relations R_1 and R_2 have the same value ($R_1.a = R_2.b$).

A concrete example from our schema is the following:

Q(name, lastname) :- dog(d_id, Name, Breed, Weight, Medication, o_id), owner(o_id, Firstname, LastName, Address, Bank, Telephone)

2.2 Complexity of Evaluating CQs

Despite their simple structure, evaluating CQs is computationally expensive. In 1977, it was proven that evaluating CQs is an NP-complete problem [CM77]. That means even though verifying a solution is possible in polynomial time, *finding* a viable solution takes exponential time w.r.t the query size in the worst case. However, research was able to identify subclasses of CQ, namely ACQs, that can be evaluated in linear time $\tilde{O}(\mid D \mid * \mid Q \mid + \mid Q(D) \mid)$, where |D| is the size of the input data, |Q| is the size of the query tree and |Q(D)| is the size of the final output [Yan81]. A small possible logarithmic factor can also be part of $\tilde{O}$. We will look at some of these methods in Section 3.4, Section 4.1 Section 4.2 and Section 4.3

CHAPTER 3

Hypergraphs and Acyclicity

Graphs are an important tool for modeling data. Due to their structure, they are able to bridge the gap between machine-readable data and human-understandable visual representation of data and its interconnections to other data. Furthermore, complex graph operations like shortest-path and minimum spanning tree (MST) can be executed on a graph structure, which is beneficial for optimization problems. In this chapter, we will introduce a generalized kind of graph - the hypergraph - and explain how query evaluation can benefit from such a structure. Afterwards, we will look into the connection between acyclicity, hypergraphs and join trees and finally introduce Yannakakis' Algorithm [Yan81] for evaluation ACQs.

3.1 Hypergraphs

We will introduce the concept of a hypergraph by firstly giving the needed definitions of a general graph, a path in a general graph, a multigraph and an (un)directed graph. The following slightly modified textbook definitions are taken from [GTH$^+$20].

3.1.1 General Definitions

Definition 1. *A **graph** G is a pair (V, E), where V is a set of vertices, and E is a set of edges. Each edge e ϵ E is an unordered pair {u, v}, where u, v ϵ V. The **size** of a graph can either be defined by the cardinality of E [GTH$^+$20] or the combined cardinality of V and E [GLPN93]*

For example, Figure 3.1 (left), shows a graph with set of vertices V = A, B, C, D, E and set of edges $E = e_1, e_2, e_3, e_4, e_5$, where $e_1 = A, B, e_2 = B, C, e_3 = C, D, e_4 = A, C, e_5 = D, E$ and $e_6 = A, E$ [GTH$^+$20].

Definition 2. *A **path** in a graph G = (V, E) is a sequence of edges $e_1, ..., e_n$ where each edge e_i is incident to e_i+1, for $1 \leq i < n$. A path is said to be simple if $e_i \neq e_j$ for $i, j \leq n, i \neq j$. The length of a path is the number of edges it contains. [GTH+20].*

For example, the simple path from vertice A to vertice D in the graph of Figure 3.1 (left) is (e_1, e_2, e_3). Two vertice x, y are connected if there exists a path $e_1, ..., e_n$ with x $\in e_1$ and $y \in e_n$.

Definition 3. *A graph G is a **multigraph** if multiple edges are permitted between two vertices.*

For example, Figure 3.1 (right), shows a multigraph with a set of vertices V = A, B, C, D and a set of edges $E = \{e_1, e_2, e_3, e_4, e_5\}$, where $e_1 = \{A, B\}, e_2 = \{B, C\}, e_3 = \{C, D\}, e_4 = \{A, C\}, e_5 = \{D, E\}$ and $e_6 = \{A, E\}$. Note that $e_7 = \{A, C\}$ and $e_8 = \{B, C\}$ are multiple edges [GTH+20].

Definition 4. *A **directed graph** DG is an ordered pair (V, E) where V is a set of vertices, and E a set of ordered pairs of vertices, called directed edges, often shown as arrows in the visual representation of DG. An edge e = (A, B) is considered directed from A to B; A is called the tail and B is called the head of the edge. A path can only follow the direction of the edges, i.e. if the only connection between A and B is a directed edge e = (A,B), then there can only be a path from A to B along e, but not the other way around [GTH+20]. An example is given in Figure 3.2*

Definition 5. *Conversely, an **undirected graph** U(V,E) is an ordered pair (V, E) with V is a set of vertices, and E a set of undirected edges, which are shown as plain lines in the visual representation of U (see Figure 3.1 (left)). If an undirected edge e = {A,B} exists between the vertices A and B, a path can follow either direction to connect the vertices A and B [GTH+20].*

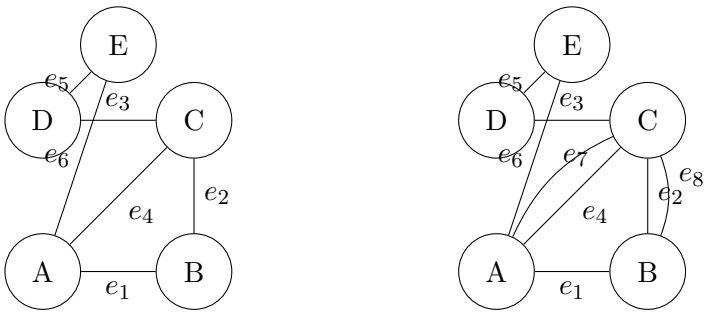


Figure 3.1: Left: A simple connected graph. Right: A multi-graph with additional edges e_7 and e_8.

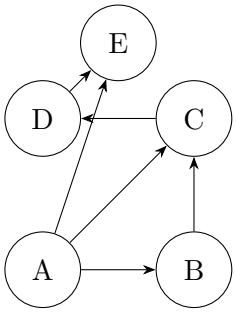


Figure 3.2: Directed graph

3.1.2 The Hypergraph

After having listed the preliminary definitions of graphs, now the concept of a hypergraph can be elaborated. Since varying definitions of a hypergraph and its properties can be found in the literature, we will compare some of them [GTH+20, BB16, GLPN93] and select the definition that fits the purpose of the thesis best.

Definition 6. *In [BB16], a **hypergraph** H(E) is defined as a set of edges $e_1, ..., e_m$, which are themselves non-empty sets of vertices; the set of vertices of H, denoted V(H), is defined as the union of all its edges. An undirected graph as defined in Section 3.1.1 is simply a special case of a hypergraph: If every edge e_i in a hypergraph has a cardinality of 2, i.e. $\mid e_i \mid = 2$, i= {1, .. . , m}, the hypergraph is an example of a standard graph [GLPN93, BB16].*

As an example, we will look at the hypergraph H given in [BB16] in Figure 3.3 (left) with the following properties: H = {a, b, c, d, e, f, g} and a = {r}, b = {r, s}, c = {s, t, u, v, w}, d = {t, u}, e = {v}, f = {x, v, w}, and g = {v, w, y, z}.

Note that we define a hypergraph H(E) as a set of edges and not as a set of edges and vertices H(V,E). Whilst the latter notation is also common, we chose the first because in a database context, all vertices are contained in edges. A single vertex would correspond to a singular attribute that is not part of a relation - a state that is not possible in a database system. Even if a relation e only contains one attribute v, it would still appear as edge $e = \{v\}$ (for more details on the connection of hypergraphs and relations, see Section 3.1.3).

Definition 7. *The **size** of a hypergraph H depends on its definition. In [GTH+20] the size $\mid H \mid$ of H is defined as the number of hyperedges, whereas each hyperedge has its own size given by the number of vertices it contains. In [BB16] and [GLPN93], a different definition of size is given: The authors define the size of a hypergraph H as the number of the sum of its edges' cardinalities, i.e. $size(H) = \Sigma_{e_i \epsilon E} \mid e_i \mid$ where E is the set of edges in H. In this work, we will use the latter definition of a hypergraph's size, since the cardinalities of relations (that is, the size of the hyperedges) do play a critical role in a database context.*

After having defined what a hypergraph is, we will now look at certain properties and elements of such a construct.

We begin with two different types of subhypergraphs.

Definition 8. *A hypergraph H' is a* ***subhypergraph*** *of a hypergraph H if H' ⊆ H; in this case we also say H' is obtained from H by removing edges, that means if we follow the definition of a hypergraph H from above as a set of its edges, H' is simply a subset of H [BB16].*

Definition 9. *A hypergraph H' is the* ***induced subhypergraph*** *of H on a set $S \subseteq V(H)$, denoted H' = H[S], if $H[S] = \{e \cap S \mid e \in H\} \setminus \{\varnothing\}$; in this case we also say H[S] is obtained from H by removing vertices (those in $V(H) \setminus S$) [BB16].*

Definition 10. *Two edges e and f of a hypergraph are called* ***properly intersecting*** *if $e \not\subseteq f$, $f \not\subseteq e$ and $e \cap f \neq \varnothing$ [GTH+20].*

For example, the two edges b = {r, s} and c = {s, t, u, v, w} in Figure 3.3 are properly intersecting.

Definition 11. *The* ***star*** *of the vertex x in the hypergraph H, denoted H(x), is defined as $H(x) = \{e \in H \mid x \in e\}$[BB16].*

As an example, the stars of the vertices in H are depicted in Figure 3.3,

Definition 12. *In a hypergraph H, an edge e is a* ***singleton edge*** *if it is of cardinality 1; by analogy, a vertex x is a* ***singleton vertex*** *if its star in H is a singleton; that is, the vertex x is contained in exactly one edge of H [BB16].*

Definition 13. *The* ***dual*** *of a hypergraph $H = e_1, ..., e_k$ on vertices $V(H) = x_1, ..., x_n$ is obtained by identifying vertices with the same star into one vertex and then "exchanging" the role of the vertices and of the edges; see Figure 3. Formally, the dual of a given hypergraph H, denoted D(H), is defined as $D(H) = \{H(x) \mid x \in V(H)\}$ [BB16]. An example for a dual is given in Figure 3.3 (right).*

3.1.3 Constructing a Hypergraph from a Query

In this section, we will look at hypergraphs in the context of a database environment, specifically we will show how a hypergraph can be constructed from a given query.

When constructing a hypergraph from a query, one does not have access to all information about the attributes of certain relations, as we only see the attributes (and relations) that are relevant to the query. If one imagines a database schema as a hypergraph (that is, edges are connected along primary and foreign key relations), the hypergraph obtained from a query is an induced subhypergraph (see Definition 9) of the schema. This holds

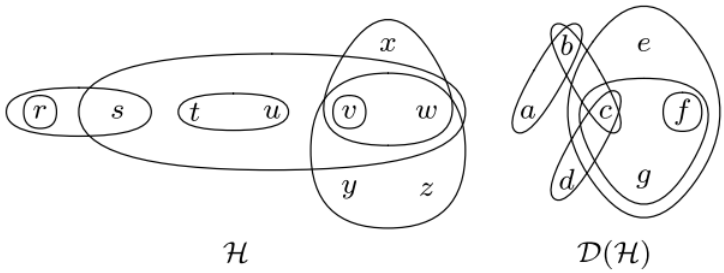


Figure 3.3: A hypergraph H = {a,b,c,d,e,f,g} (left) and its dual (right). The stars of the vertices in H are the following: H(r) = {a, b}, H(s) = {b, c}, H(t) = H(u) = {d, c}, H(v) = {e, c, f, g}.

```
SELECT DISTINCT o.Firstname, o.Lastname, o.Telephone, d.name, v.payment
FROM OWNER o
JOIN DOG d ON o.O_ID = d.O_ID
JOIN VISIT v ON o.O_ID = v.O_ID
WHERE d.Breed = 'Chihuahua'
AND v.Date = '2024-11-18';
```

Figure 3.4: Querying the database for all payments of Chihuahua owners that have been to the vet on a certain day in SQL.

```
Owner(O_ID, Firstname, Lastname, Telephone).
Dog(D_ID, O_ID, Name, Breed).
Visit(V_ID, O_ID, Payment, Date).

// Query
Result(Firstname, Lastname, Telephone, DogName, Payment) :-
    Owner(O_ID, Firstname, Lastname, Telephone),
    Dog(D_ID, O_ID, DogName, "Chihuahua"),
    Visit(V_ID, O_ID, Payment, "2024-11-18").
```

Figure 3.5: Querying the database for all payments of Chihuahua owners that have been to the vet on a certain day in datalog.

because the hypergraph of a query cannot contain any vertices that are not part of the database schema.

In the above examples (see Figure 3.4 and Figure 3.5), the hypergraph H has three edges: H = {O, D, V}. The subset of nodes that can be deducted from the query and matched to their relations are the following: O = {Firstname, Lastname, Telephone, O_ID}, D = {Name, OwnerID, D_ID, Breed} and V = {Payment, OwnerID, Date}. The resulting

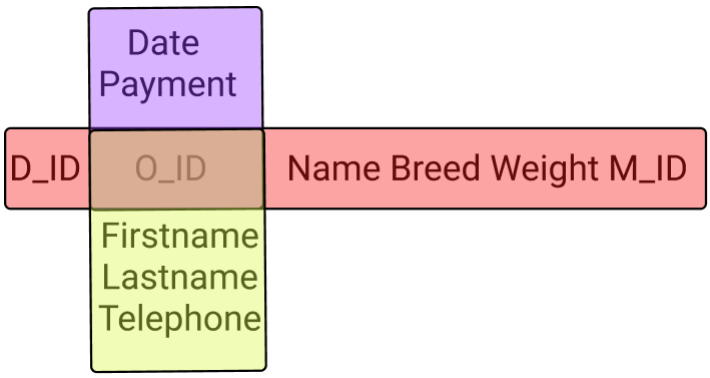


Figure 3.6: The constructed hypergraph from the query above (see 3.4)

hypergraph H can be seen in Figure 3.6. It is important to note that overlaps of edges have to be deducted from the equality conditions and not by semantics (e.g. an attribute o.ID would not be the same node as d.ID, unless both would appear on one side of an equality condition: d.ID = o.ID).

3.2 Acyclicity and Join Trees

In this section, we will introduce a subclass of CQs, namely acyclic conjunctive queries (ACQs). Commonly, it can be said that a CQ is acyclic if it has a join tree. Note that our notion of the term "acyclicity" is the so-called alpha-acyclicity [BB16, Fag83].

A tree is defined as the following:

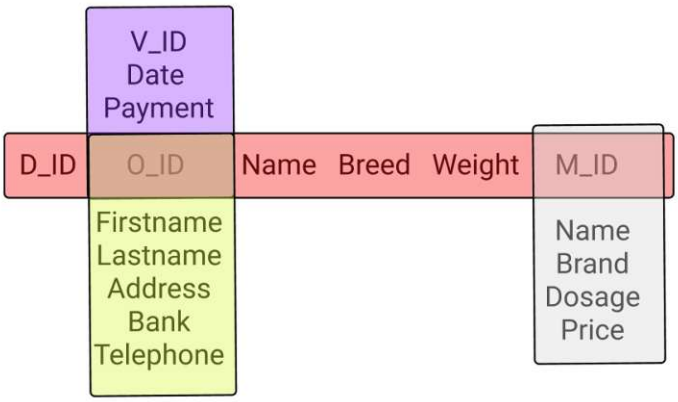
Definition 14. *A **rooted tree** T (V, E) is a connected, acyclic graph with a distinctive root node. Note that depending on the illustration of the tree, the root node can be chosen arbitrarily (see Figure 3.7).*

A join tree is a specific variation of a general tree:

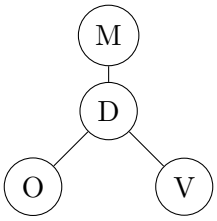
Definition 15. *A **join tree** T, is a rooted tree with root r that has an additional labeling function λ: $\langle T, r, \lambda \rangle$. λ is a bijection that assigns to each node of T one of the relations $R_1, ..., R_n$ that are part of a query while also fulfilling the connectedness condition: if some attribute A_1 occurs in both relations n_i) and $\lambda(n_j)$ for two nodes n_i and n_j, then A_1 has to occur in the every node (and thus relation) along the path between n_i and n_j [GLL+23a].*

Note that a hypergraph can have many possible join trees. Examples can be seen in Figure 3.7.

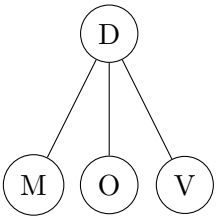
Theorem 1. *A hypergraph H(V, E) is α-acyclic if it has a join tree [GLL+23a]. A detailed proof of this theorem can be found in [Fag83].*



(a) The hypergraph of a query containing all relations of the schema.



(b) Join tree of the hypergraph above.



(c) Join tree of the hypergraph above.

Figure 3.7: The hypergraph of the a query containing the relations {Dog, Owner, Visit, Medication} and two possible join trees with M = Medication, D = Dog, O = Owner and V = Visit

To decide whether a CQ is acyclic can be done by applying the GYO-algorithm (see Section 3.3). In the positive case, a join tree is constructed [GLL+23a].

3.3 GYO-Reduction

To check whether a hypergraph of a CQ is acyclic, one can apply the GYO-algorithm, or GYO-Reduction. The algorithm was originally developed by Graham, Yu and Ozsoyoglu [YO79, Gra79] in 1979. The procedure can be used to check for acyclicity due to the design of the algorithm: If a hypergraph can be reduced to an empty set of edges, it is acyclic. While removing edges, a join tree can be built in parallel, using the intermediate

results (i.e. removed edges) of the reduction. The pseudo-code of the algorithm can be read in Algorithm 3.1

Algorithm 3.1: GYO-reduction [YO79, Gra79]

Input: A hypergraph $H(V, E)$
Output: Reduced set of edges E

```
GYO-REDUCTION H(V, E) while possible do
    if ∃e ∈ E : ∀v ∈ e : ∄e' ∈ E \ {e} : v ∈ e' then
        delete e;
    end
    if ∃e ∈ E : ∃w ∈ E \ {e} : ∀v ∈ e : (v ∈ w ∨ ∄e' ∈ E \ {e} : v ∈ e') then
        assign w as witness of e and delete e;
    end
end
return E;
```

Definition 16. *If the GYO-reduction is applied to a hypergraph, a hyperedge e can be removed if there exists another hyperedge f that has the following properties: for every vertex x that does not appear exclusively in e, it must be that $x \in f$. That means if e shares vertices with other edges, there must exists one edge f that contains all of these shared nodes. Edge f is then called a* ***witness*** *of e. A vertex that is contained in e but not in any other edge is a so-called ear-node or ear-vertex [YO79, Gra79].*

According to the definition above, every edge that shares no vertices with another edge can be trivially removed, as can any edge that is completely contained within another edge. Note that there can be cases of unconnected hypergraphs, in which case the hypergraph does not have a join tree, but a join forest.

During the reduction, the following two steps are executed until no more edges eligible for removal are found (or all edges have already been removed):

1. Find and select an edge e of the hypergraph H that has either a) a witness e' and contains one or more ear-vertices or b) is completely contained within another edge or c) shares no vertices with another edge.

2. Remove e from H, i.e. remove e from the set of hyperedges.

As an example for how to decide whether a hypergraph is acyclic and how to build one or more join trees from a hypergraph using the GYO-reduction, we will revisit the examples given in Figure 3.7. While executing the GYO-reduction, a join tree is created.

To obtain the join tree in Figure 3.7b from the hypergraph given in Figure 3.7a, we choose edge O first and remove it with edge D as its witness, which also means that we put down D as parent node of O. Next, we remove edge V, which has D as parent

as well. Finally, we choose edge D to be removed, making M its witness and parent. Finally, M can be trivially removed, because it has no more shared nodes. Since the algorithm reduces the set of edges to an empty set, the hypertree is acyclic.

In Figure 3.7c, the algorithm starts by picking another edge as the first to be removed. It removes M first, then O and finally V, all having D as witness and thus parent node. Finally, D can also be removed and the result is again the empty set.

Before closing this section, we want to highlight one interesting property of acyclic hypergraphs: an acyclic hypergraph can have a cyclic subgraph. This is illustrated by the examples given in Figure 3.8 and Figure 3.9. If we process the hypergraph given in Figure 3.8, we find that it is acyclic:

1. W.l.o.g, we identify B as first ear-vertex and consequently remove edge {A,B,C} since edge {A,C,E} acts as a witness.
2. Next, we identify D as ear-vertex and consequently remove edge {E,D,C}. Again,edge {A,C,E} acts as a witness.
3. Next, we identify F as ear-vertex and consequently remove edge {E,F,A} with edge {A,C,E} as witness.
4. All nodes of the remaining edge {A,C,E} are now ear-vertices; the edge can be removed.
5. All edges have been removed by the GYO-Algorithm, which means that the hypergraph is acyclic.

Interestingly, if we try to apply the same procedure to the hypergraph in Figure 3.9, which is a subgraph of Figure 3.8, we find that the GYO-Algorithm fails: There is no edge is fully contained in another edge. There is no edge that shares no vertices with another edge. An also, there is no edge that has an ear-vertex and a witness. The algorithm fails, the hypergraph is cyclic [Fag83].

3.4 Yannakakis' Algorithm

Once we have checked whether a hypergraph of a query is acyclic and have created a join tree, we can apply Yannakakis' algorithm [Yan81, TGR22, Pic23] to evaluate it without creating intermediate results that contain tuples that are not part of the final result - the so-called "dangling tuples".

This is accomplished by traversing the join tree up and down while applying semi-joins that eventually reduce all relations to the tuples that are part of the end result. In the final step, only these remaining tuples have to be joined, meaning that the final joins are as minimal as possible for a given join tree, since after every join, the intermediate result will only contain tuples that are part of the final result.

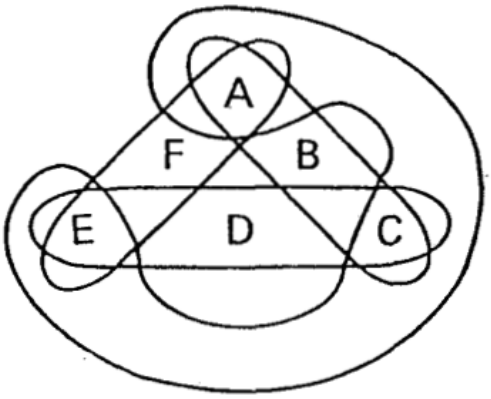


Figure 3.8: An acyclic hypergraph [Fag83]

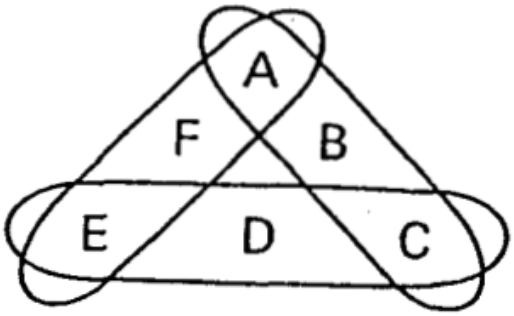


Figure 3.9: A cyclic hypergraph [Fag83]

A semi-join is defined as follows:

Definition 17. $R \ltimes S := \{r | r \in R \wedge \exists s \in S \wedge r_{[A_1,\ldots,A_n]} = s_{[A_1,\ldots,A_n]}\}$. *This means that the semi-join returns all tuples from R that have a join partner in relation S.*

The algorithm consists of three steps:

1. A bottom-up traversal in which every node n is semi-joined with its child-nodes c_i: $n \ltimes c_i$. This removes all tuples from n that are not present in c_i (upwards propagation).

2. A top-down traversal that has the opposite join-direction of step 1: $c_i \ltimes n$. This removes all tuples from c_i that are not present in n (downwards propagation).

3. The remaining tuples are joined according to the join conditions. This last step can be done either bottom-up or top-down [Pic23, Yan81].

Notably, the last step does not need to be done in a separate traversal. Since the relations are joined iteratively, it could also be done during step 2, so that the final result is built incrementally [TGR22].

Yannakakis' algorithm works because it efficiently exploits the structural properties of acyclic conjunctive queries (ACQs) and their associated join trees. The first bottom-up traversal performs semi-joins, which filter tuples in the parent relations by ensuring only those tuples are kept, which appear in the child nodes. This way, no information that is needed later on will be lost. The top-down traversal further refines these results by eliminating all tuples in the child nodes that are not contained in the parent relations. After the second step, all dangling tuples have been eliminated and the actual computation produces minimal intermediate results for a given join tree [Yan81, GLL+23a].

Correctness Proof. The following is a proof sketch of Yannakakis' algorithm as shown in [GLL+23b].
Assume a join tree T. For every node $t \in V(T)$ let T_t be the subtree of T that has t as its root node. Furthermore, let R_t be the relation at node t. Regarding the traversals, let R'_t, R''_t and R'''_t denote the result of the first / second / third traversal of the join tree. Formally, R'_t, R''_t and R'''_t can be defined the following way:

After the first bottom-up traversal, the following holds: $R'_t = \pi_{vars(t)}(\bowtie_{v \in V(T_t)} R_v)$ for each $t \in T$. This means that for any subtree T_t, all relations of T_t are joined and the results are projected onto the attributes of t. Intuitively, this means that the result consists of all tuples in t that have join partners with its child nodes. This concludes the upwards propagation.

After the top-down traversal, the following holds: $R''_t = \pi_{vars(t)}(\bowtie_{v \in V(T)} R_v)$ for each $t \in T$. After the completed upward propagation (R'_t), this step ensures that dangling tuples are removed from child nodes by joining all tuples and then projecting to the attributes of t. This concludes the downward propagation.

After the second bottom-up traversal, the following holds: $R'''_t = \pi_{vars(T_t)}(\bowtie_{v \in V(T)} R_v)$ for each $t \in T$. This step joins all relations in T sans the dangling tuples that have been removed by prior steps. After completing this process for all nodes in the join tree, the root node now contains all results of the query. As mentioned above, this step can be incorporated into step 2 as a means of optimization.

Complexity. Now it becomes apparent why the algorithm evaluates in linear time with respect to the size of the input D, the quer size Q and the size of the final result $Q(D)$: $\tilde{O}(\mid D \mid * \mid Q \mid + \mid Q(D) \mid)$. In the first two steps, the semi-joins, every tuple of the input relations has to be look for a join partner within its neighboring relations in the join tree. The bigger the size of the input, the more time consuming this step will be. And the size of the final result r is directly connected to the third step: the more tuples remain, the longer it takes to join the relations.

In the following sections, we will execute Yannakakis' Algorithm step-by-step on the query below (see Figure 3.10), with the corresponding join tree depicted in Figure 3.11.

```
SELECT *
FROM Medication M, Owner O, Dog D, Visit V
WHERE M.name = D.Medication
AND D.OwnerID = O.ID
AND D.ID = V.DogID
AND O.ID = V.OwnerID
```

Figure 3.10: A simple CQ.

The query simply joins the relations MEDICATION, OWNER, DOG and VISIT and returns all "active" records, e.g. if a medication is currently prescribed to no one, it will not appear in the final result.

3.4.1 Bottom-Up Traversal

In the bottom-up traversal, the following semi-joins are computed: $Owner \rtimes Dog$, $Visit \rtimes Dog$ and $Dog \rtimes Medication$.

Note that the first semi-join (Figure 3.12) in the bottom-up traversal causes no tuple reduction, since there is no tuple in DOG that has no join partner in OWNER.

The second semi-join (Figure 3.13) in the bottom-up traversal removes the last row in DOG, since there is no visit in VISIT of a Dog 6 with Owner 101.

The third semi-join (Figure 3.14) in the bottom-up traversal removes the last three medications in MEDICATION, since they are not prescribed to any dog in DOG.

3.4.2 Top-Down Traversal

In the top-down traversal, the following semi-joins are computed: $Medication \rtimes Dog$, $Dog \rtimes Owner$ and $Dog \rtimes Visit$.

The first semi-join (Figure 3.15) in the top-down traversal removes no rows, because there is no medication present in DOG that is not present in MEDICATION.

The second semi-join (Figure 3.16) in the top-down traversal removes owner 105 from OWNER, because they do not own any dog in the relation DOG.

The last semi-join (Figure 3.17) in the top-down traversal removes visit 5 from VISIT, since no such Dog/Owner combination is present in DOG.

3.4.3 Final Traversal

What is left to do is joining all the tuples and returning the output result. For spatial reasons, we will apply a projection onto the result and only select the columns M.name, D.name., O.lastname V.payment and V.date. The result can be seen in Table 3.1.

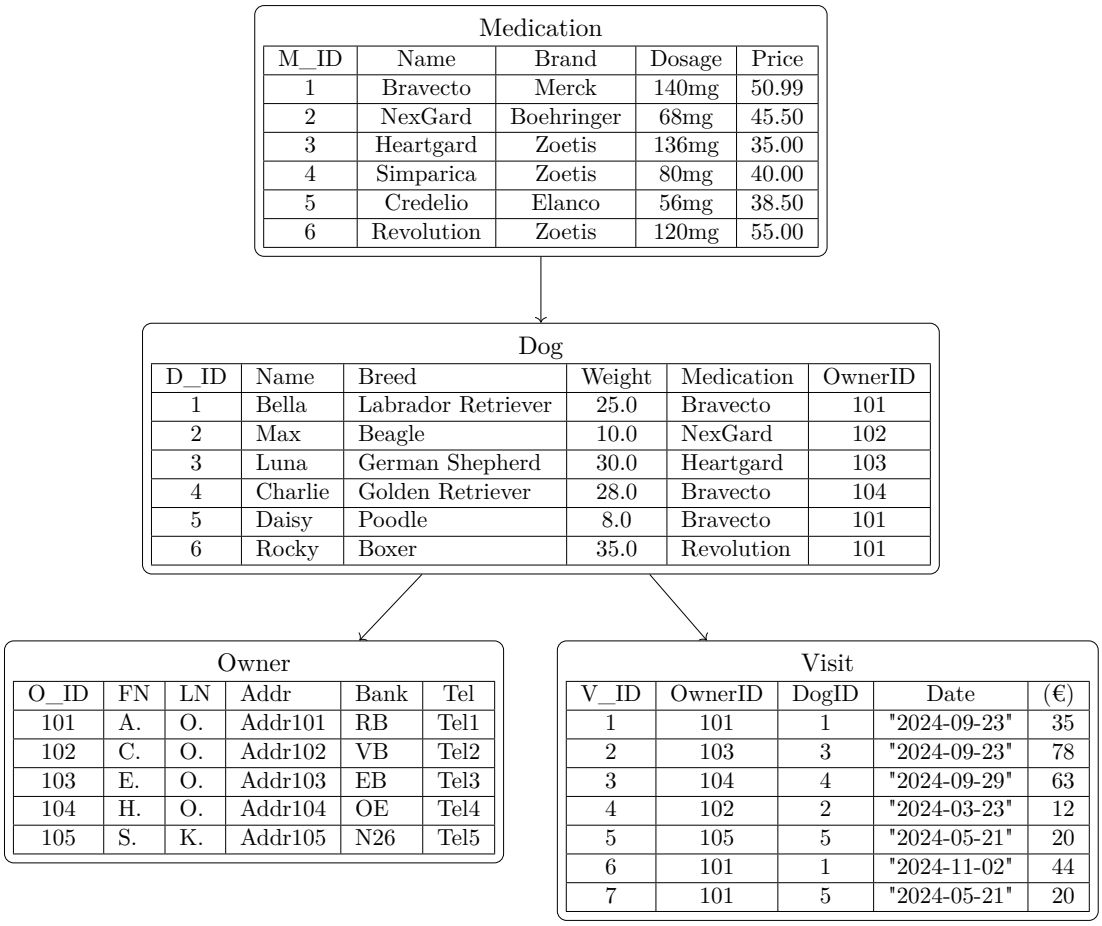
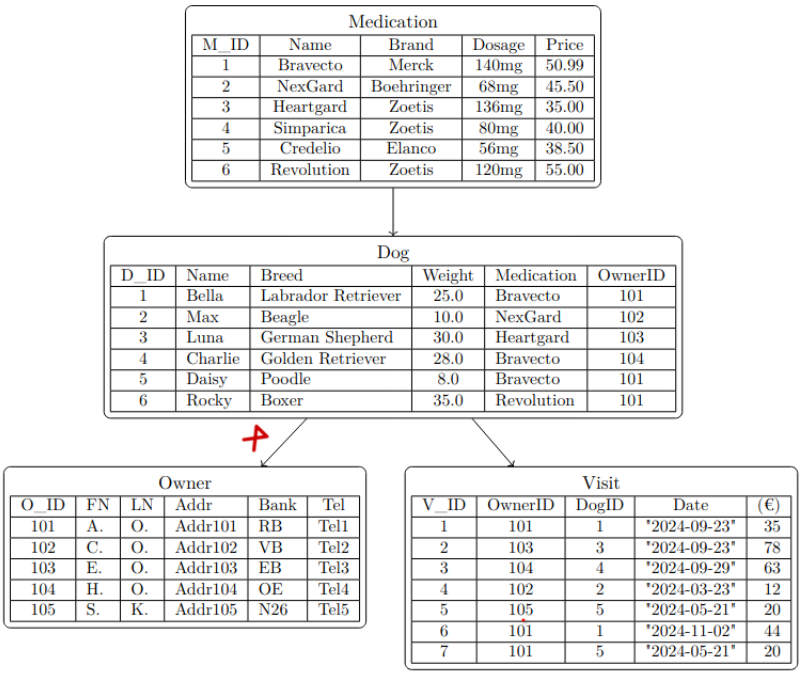


Figure 3.11: A join tree for a query that contains the relations Owner O, Medication M, Visit V, and Dog D.

M.name	**D.name**	**O.firstname**	**v.payment**	**v.date**
Bravecto	Bella	A.	35	2024-09-23
NexGard	Max	C.	12	2024-03-23
Heartgard	Luna	E.	78	2024-09-23
Bravecto	Charlie	H.	63	2024-09-29
Bravecto	Daisy	A.	20	2024-11-02
Bravecto	Bella	A.	44	2024-11-02

Table 3.1: The final result of the join, projection applied.



Medication

M_ID	Name	Brand	Dosage	Price
1	Bravecto	Merck	140mg	50.99
2	NexGard	Boehringer	68mg	45.50
3	Heartgard	Zoetis	136mg	35.00
4	Simparica	Zoetis	80mg	40.00
5	Credelio	Elanco	56mg	38.50
6	Revolution	Zoetis	120mg	55.00

Dog

D_ID	Name	Breed	Weight	Medication	OwnerID
1	Bella	Labrador Retriever	25.0	Bravecto	101
2	Max	Beagle	10.0	NexGard	102
3	Luna	German Shepherd	30.0	Heartgard	103
4	Charlie	Golden Retriever	28.0	Bravecto	104
5	Daisy	Poodle	8.0	Bravecto	101
6	Rocky	Boxer	35.0	Revolution	101

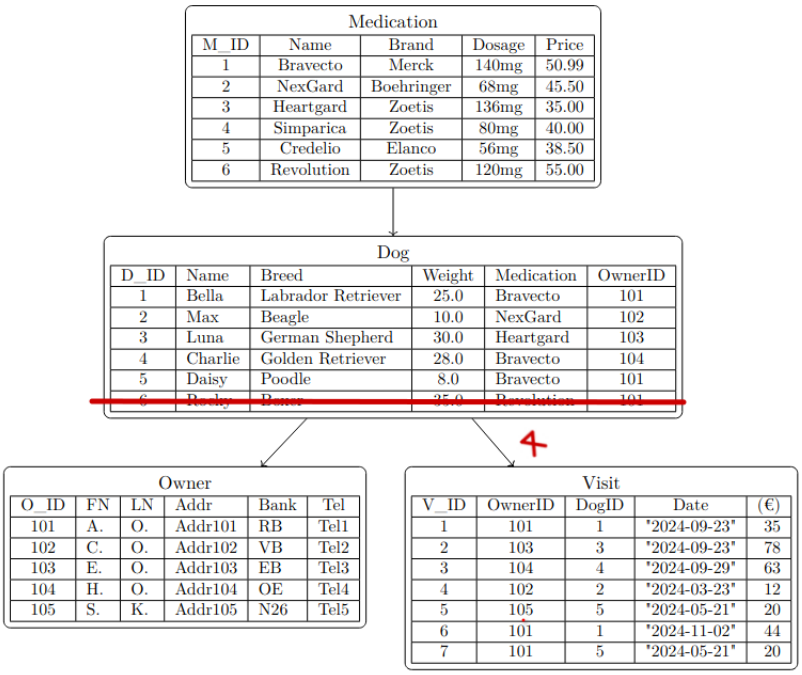
Owner

O_ID	FN	LN	Addr	Bank	Tel
101	A.	O.	Addr101	RB	Tel1
102	C.	O.	Addr102	VB	Tel2
103	E.	O.	Addr103	EB	Tel3
104	H.	O.	Addr104	OE	Tel4
105	S.	K.	Addr105	N26	Tel5

Visit

V_ID	OwnerID	DogID	Date	(€)
1	101	1	"2024-09-23"	35
2	103	3	"2024-09-23"	78
3	104	4	"2024-09-29"	63
4	102	2	"2024-03-23"	12
5	105	5	"2024-05-21"	20
6	101	1	"2024-11-02"	44
7	101	5	"2024-05-21"	20

Figure 3.12: *Owner* ⋊ *Dog*



Medication

M_ID	Name	Brand	Dosage	Price
1	Bravecto	Merck	140mg	50.99
2	NexGard	Boehringer	68mg	45.50
3	Heartgard	Zoetis	136mg	35.00
4	Simparica	Zoetis	80mg	40.00
5	Credelio	Elanco	56mg	38.50
6	Revolution	Zoetis	120mg	55.00

Dog

D_ID	Name	Breed	Weight	Medication	OwnerID
1	Bella	Labrador Retriever	25.0	Bravecto	101
2	Max	Beagle	10.0	NexGard	102
3	Luna	German Shepherd	30.0	Heartgard	103
4	Charlie	Golden Retriever	28.0	Bravecto	104
5	Daisy	Poodle	8.0	Bravecto	101
6	Rocky	Boxer	35.0	Revolution	101

Owner

O_ID	FN	LN	Addr	Bank	Tel
101	A.	O.	Addr101	RB	Tel1
102	C.	O.	Addr102	VB	Tel2
103	E.	O.	Addr103	EB	Tel3
104	H.	O.	Addr104	OE	Tel4
105	S.	K.	Addr105	N26	Tel5

Visit

V_ID	OwnerID	DogID	Date	(€)
1	101	1	"2024-09-23"	35
2	103	3	"2024-09-23"	78
3	104	4	"2024-09-29"	63
4	102	2	"2024-03-23"	12
5	105	5	"2024-05-21"	20
6	101	1	"2024-11-02"	44
7	101	5	"2024-05-21"	20

Figure 3.13: *Visit* ⋊ *Dog*

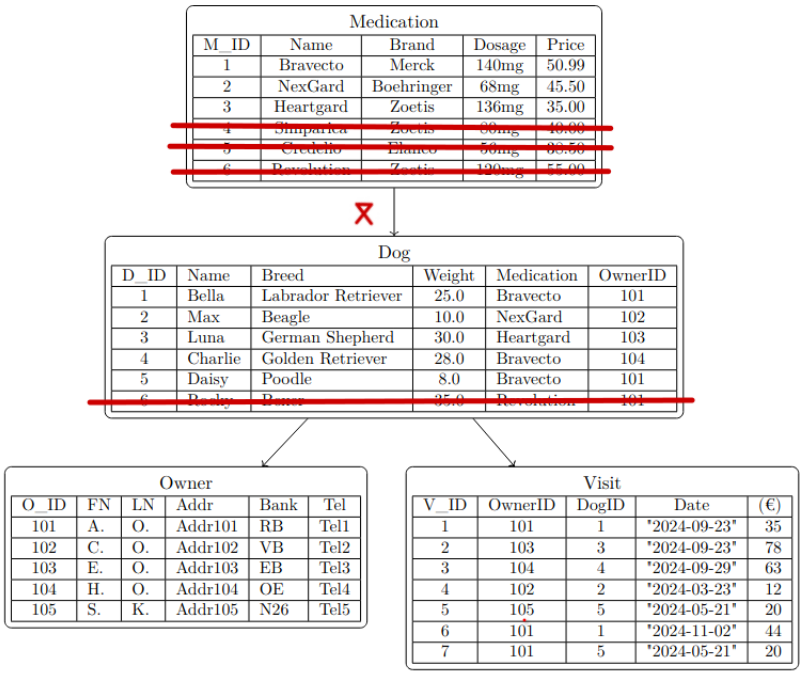


Figure 3.14: $Dog \rtimes Medication$

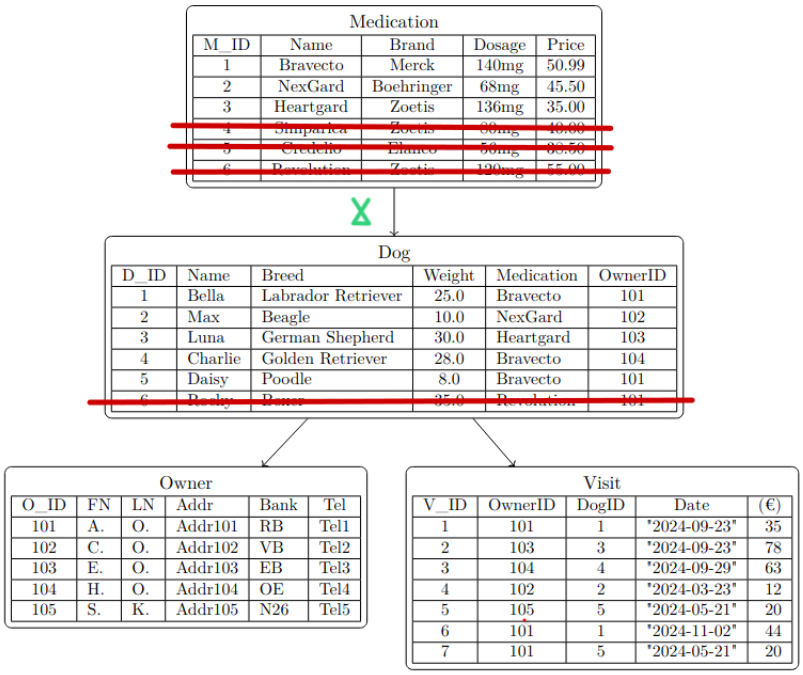


Figure 3.15: $Medication \rtimes Dog$

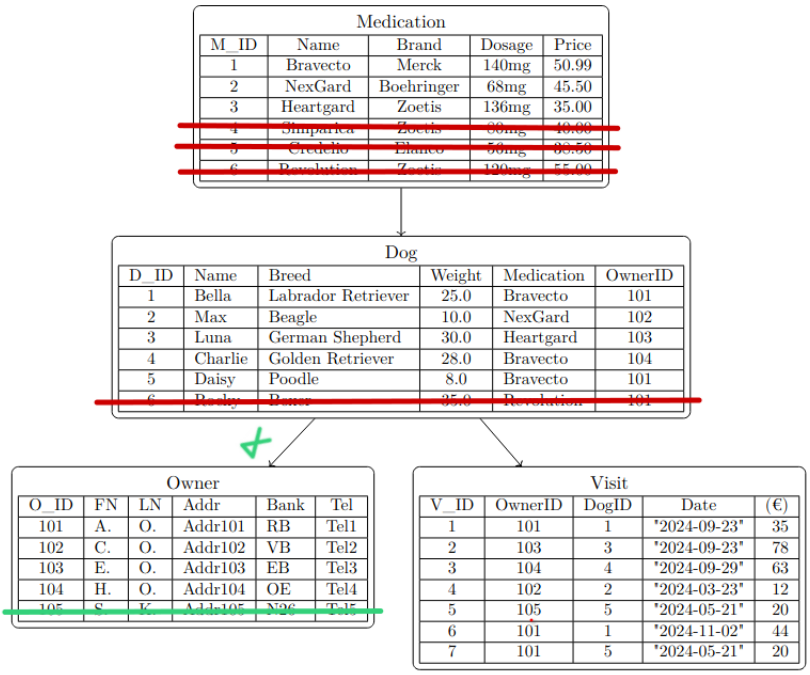


Figure 3.16: $Dog \rtimes Owner$

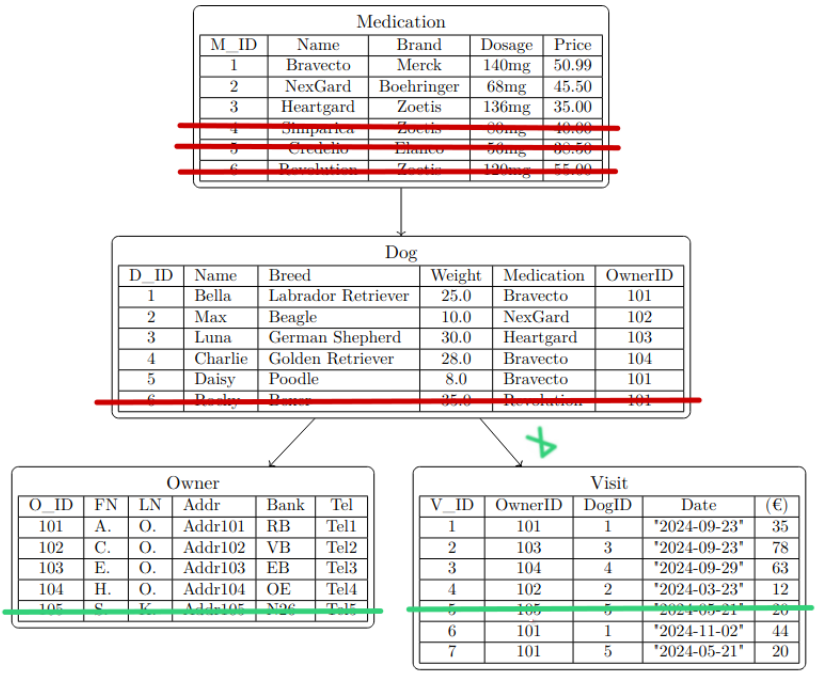


Figure 3.17: $Dog \rtimes Visit$

CHAPTER 4

Beyond ACQs: 0MA and Guarded Aggregate Queries

With his algorithm, Yannakakis [Yan81] introduced a method to eliminate tuples that are not part of the final result (dangling tuples) from the final join and thus greatly reduced the size of intermediate results. However, there is still one quite computationally expensive action to take: joining the reduced intermediate results. As research [GLS01, GLL+23a, LPS24] has shown, Yannakakis' approach can be expanded and tweaked to reduce costs even more. In this chapter, we will take a look beyond ACQs and talk about Boolean ACQs (Section 4.1), 0MA queries (Section 4.1), guarded aggregate queries (Section 4.3 and piecewise-guarded aggregate queries (Section 4.4).

4.1 Boolean ACQs

Boolean ACQs are a class of simple queries that only check for the existence of a result, i.e. if the set of answer tuples is non-empty, the result is true, otherwise it is false. For this class of queries, it has been formally proven [Yan81, GLS01] that one bottom-up traversal of semi-joins suffices to evaluate the query. This also becomes apparent in the example and the correctness proof given in Section 3.4: If the bottom-up semi-joins are done and the root node is non-empty, we can already conclude that *some* values will be returned, since the result can never be less than the size of the root node after the bottom-up traversal, it only grows by joining other relations [TGR22, GLS01]. Naturally, this speeds up the evaluation greatly, since the most expensive action - the final joining of relations - does not have to be performed.

4.2 0MA Queries

Following up on Boolean ACQs, the authors of [GLL+23a] have introduced a whole class of queries that can be evaluated in one bottom-up traversal of the join tree and more importantly without any materialization: the 0MA query class (short for zero materialization answerable queries).

In their paper [GLL+23a], the authors give the following example:

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

Since there are only two relations in this query, it is trivially acyclic and the bottom up traversal consists of only one semi-join. As we recall, in the bottom-up traversal, the child node restricts the parent node. Here, the semi-join $COURSES \ltimes EXAMS$ would remove any tuple from exams that does not match the *cid* of biology courses in COURSES. As we need no attributes from the COURSES relation, all that is left is to apply the grouping and aggregation. The result is obtained without joining any relations [GLL+23a]. This also follows from the correctness proof in Section 3.4.

Formally, a 0MA query Q has to fulfill three properties:

1. Q has to be in **aggregation normal form** $\gamma_U(\pi_S(Q'))$. That means additionally to Q', which can only contain natural joins and selection, the grouping operator γ and aggregations are allowed.
2. Q has to be **guarded**. The definition of guardedness in [GLL+23a] is the following:

   Definition 18. *Guardedness: All attributes $G_1, ..., G_i$ appearing in a GROUP BY statement (aggregated or not) and aggregates $A_1, ..., A_i$ have to be in the same relation R of Q. R is then called the "guard" of Q.*

3. Q has to be **set-safe**. This means that duplicate elimination δ before the aggregation of attributes has no effect on the final result, i.e. $\gamma_U(\pi_S(Q'))=\gamma_U(\delta(\pi_S(Q')))$

0MA queries offered a good means of optimization for guarded queries that contain the aggregate functions MIN, MAX and COUNT DISTINCT. But even though many queries fulfill the above properties, guardedness and set-safety are quite restrictive properties. Set-safety is especially restrictive, since aggregations like SUM, AVG, MEDIAN and COUNT(*) are per definition excluded.

The authors admit this limitation, but also note that even if a query is not 0MA, there is potential for cost reduction: if a subtree of the join tree is 0MA, then some joins can still be omitted and overall the materialization of intermediate results is reduced [GLL+23a].

4.3 Guarded Aggregate Queries and their Evaluation

The introduction of 0MA queries offered a new perspective, but as written above, its benefits are only applicable to a small percentage of queries that are both guarded and set-safe. In [LPS24], the authors were able to drop the set-safety restriction and introduced "guarded aggregate queries". By developing a new operator that accounts for the frequency of the appearance of each tuple (the duplicates) and propagating this information further up during the bottom-up traversal, they were able to evaluate guarded aggregate queries with zero materialization.

4.3.1 Frequency Propagation in Guarded Aggregate Queries

Definition 19. *Let Q be a query of the form $\gamma[g_1, ...g_n][A_1(a_1), ...A_j(a_m)](R1 \bowtie R2... \bowtie R_o)$. γ stands for GROUP BY, A_j for aggregation. We call Q a guarded aggregate query (or simply, "guarded query"), if $(R_1 \bowtie R_2... \bowtie R_o)$ 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 [LPS24].*

If the set-safety is dropped, how can duplicate tuples be accounted for? The main idea is to note the frequency, i.e. the number of duplicates of every tuple as an extra attribute in each tuple and propagate this one additional attribute upwards the join tree. The initialization is formalized as $Freq(u) = R(u) \times \{(1)\}$, where u is a node in the join tree and $R(u)$ is the corresponding relation. $Freq(u)$ adds one additional column to $R(u)$, saving the frequency of each row.

The propagation of frequencies can be formalized in three statements:

- $Freq_0(u) := R(u) \times \{(1)\}$
- $Freq_i(u) := \gamma[Att(u), c_u^i \leftarrow SUM(c_u^{i-1} * c_{u_i})](Freq_{i-1}(u) \bowtie Freq(u_i))$
- $Freq(u) := \rho_{c_u \leftarrow c_u^k}(Freq_k(u))$

For better understanding, we will include the example given in [LPS24]. Consider the following query and its corresponding join tree in Figure 4.1:

```
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')
```

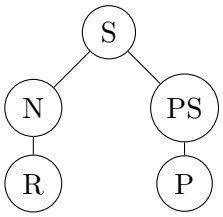


Figure 4.1: The corresponding join tree of the query above. S = supplier, N = nation, R = region, PS = partsupp, P = part

For this query, the result after one bottom-up traversal with applied frequency propagation can be found in Figure 4.2. As can be seen, the frequency attribute c for the first record in supplier is 30. How is this computed? In the first step the count of each tuple c is initialized to 1 Since REGION and PART are leaves, these are already their final c-values. This corresponds to the formal statement $Freq_0(u) := R(u) \times \{(1)\}$. The step from PART to PARTSUPPLIER or REGION to NATION, respectively, denotes the frequencies of the R and P tuples in the child nodes and thus completes $Freq_i(u) := \gamma[Att(u), c_u^i \leftarrow SUM(c_u^{i-1} * c_{ui})](Freq_{i-1}(u) \bowtie Freq(u_i))$. For example, the first c-value in NATION is calculated by summing up the counts of r_1 in REGION and multiplying each count by 1, the initialized value in NATION: $1 * 1 + 1 * 1 + 1 * 1$ The step from NATION and PARTSUPPLIER to SUPPLIER involves the summation and multiplication of the frequencies propagated in the subtrees. We have to look at all appearances of n_1 in NATION and s_1 in PARTSUPPLIER. In PARTSUPPLIER, this means we have to get the frequency of the tuples (s_1, p_1), (s_1, p_2) and (s_1, p_3). For (s_1, p_1) the frequency is 3. This is now multiplied with all frequencies of n_1 in NATION: $3*3+3*2 = 15$. For (s_1, p_2) the frequency is 2. This is now also multiplied with all frequencies of n_1 in NATION: $2 * 3 + 2 * 2 = 10$. For (s_1, p_3) the frequency is 1. Multiplied with all frequencies of n_1 in NATION we get $1 * 3 + 1 * 2 = 5$. And $15 + 10 + 5 = 30$

At the end of the bottom-up traversal, the root relation, which acts as a guard to the query, holds all aggregation and grouping attributes and the frequency of each tuple which reflects the size of the final result if all the joins have been realized. This allows the system to compute aggregations which are not set-safe, as they can be rewritten and expressed with frequency attributes. In the following list, B denotes an attribute of guard R and c_r the frequency attribute [LPS24].

- COUNT() $\rightarrow$ SUM(c_r)
- COUNT(B) $\rightarrow$ SUM(IF(ISNULL(B), 0, c_r))

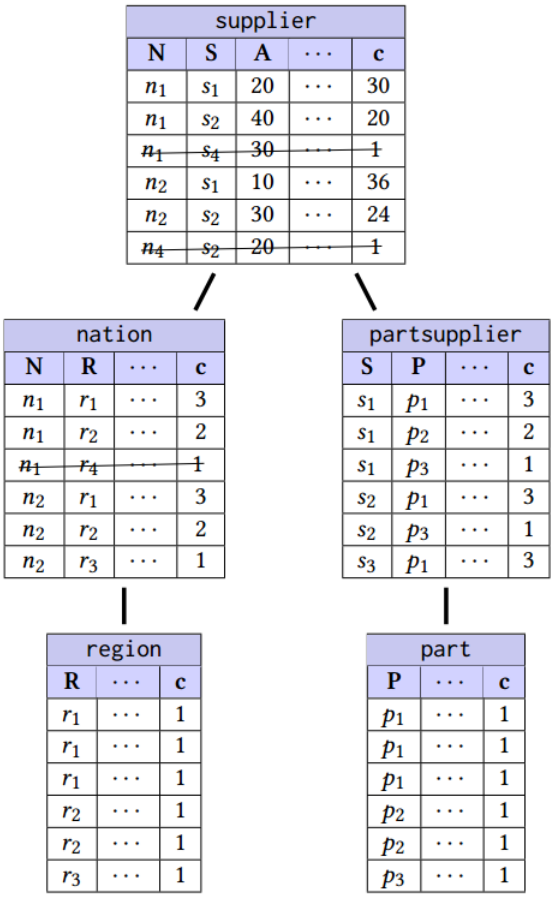


Figure 4.2: Evaluation of the query from join tree 4.1 [LPS24]

- SUM(B) → SUM(B*c_r)
- AVG(B) → SUM(B*c_r)/COUNT(B)
- MEDIAN(B) → PERCENTILE(0.5, B, c_r)

In the initial version of [LPS24], the optimization they offered was based only on guarded aggregate queries. And even though it showed great speed-gains, it was only applicable to a very narrow subset of queries, as was seen in the benchmark of the same paper: While the optimization was applicable to all queries in the STATS-CEB dataset [HWW+21] and the SNAP dataset [LK14], only 5 out of 33 queries in the JOB dataset [LGM+15] were eligible for optimization, in LSQB [MLK+21] only 2 out of 9 and in TPC-H [TPCb] only 2 out of 22.

4.4 Piecewise-guarded Aggregate Queries

The experimental evaluation in [LPS24] has shown, that guarded aggregate queries are able to cover a bigger percentage of queries in the benchmarks used to evaluate them than 0MA. Still, in real life scenarios, the guardedness conditions proves too restrictive. It is simply too often not the case that all aggregated attributes and on top of that all grouping attributes come from the same relation. The authors were able to relax the guardedness restriction for queries containing MIN, MAX, SUM, COUNT, and AVG and introduced a new term: *piecewise-guarded aggregate queries.*

Definition 20. *Let Q be a query of the form $\gamma[g_1, ...g_n][A_1(a_1), ...A_j(a_m)](R1 \bowtie R2... \bowtie R_o)$. Again, γ stands for GROUP BY, A_j for aggregation. We call Q a piecewise-guarded aggregate query (or simply, "piecewise-guarded query"), if $(R_1 \bowtie ... \bowtie R_o)$ is acyclic and there exists a relationR_{i_0} that contains all grouping attributes and, for every $j \in \{1, . . . ,m\}$, 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 $A_j(a_j)$ [LPS24].*

R_0, the guard of the attributes in the GROUP BY clause acts as so-called root guard, i.e. this is the root of the join tree. The relations guarding one or more aggregates are simply referred to as guards [LPS24].

Evaluating a query with aggregates that are all contained in the relation of the root node works exactly as described in Section 4.3.1. But what about queries that contain aggregated attributes that are not guarded by the root guard? As stated before, the relaxation is only valid for MIN, MAX, SUM, COUNT, and AVG. The idea is simple: those aggregates that are needed in the final result get propagated upwards along with their frequency [LPS24].

In the revision of [LPS24], the authors introduced AggJoin (see Section 4.5), a new operator in Spark that provides optimization not only for guarded queries, but also for piecewise-guarded queries. With this improvement it was applicable to a wider subset of queries: The new optimization was applicable to all queries in the STATS-CEB dataset [HWW+21] and the SNAP dataset [LK14]. But this time, all 113 JOB queries, 7 out of 22 TPC-H queries , 30 out of 99 TPC-DS queries and 2 out of 9 LSQB queries could be processed with AggJoin. Notably, not all queries in the benchmarks are aggregate queries. For more details, see Figure 6.4.

In the next section, we will take a closer look at AggJoin in theory and in practical terms.

4.5 AggJoin

One example of technology that takes advantage of the above described notions of *guardedness* and *piecewise-guardedness* is the so-called *AggJoin* extension of Spark (see Chapter 5), that was first introduced in [LPS24]. The implementation and a benchmark environment are publicly available at https://github.com/dbai-tuw/spark-eval.

Looking back at Figure 4.2, one can see that the algorithm works because the guard of the query, here SUPPLIER, already holds all attributes that have to be aggregated eventually. We can avoid materializing intermediate results by propagating frequencies because we know that the guard will contain the needed columns and values. But what if this is not the case?

Logical Optimization

Assume that an aggregate $A_j(a_j)$ is part of a query to be evaluated, but not guarded by the root guard r. Furthermore assume that $A_j \in \{MIN, MAX, SUM, COUNT, AVG\}$ and that a_j is of the form $f_j(\overline{B}_j)$, where f_j is a function on the attributes A'_j, e.g. $A_j(a_j) = MAX(a_i + a_j)$. We will now search for the highest node w in the join tree T_u that contains all attributes $\overline{B}_j$. Note that $w \neq r$ since the query is piecewise-guarded, but not guarded. To pass $A_j(a_j)$ from w to r, we have to add an additional attribute called Agg_j to every node u on the path from w to r. Formally, this processed can be described the following:

$\gamma[Att(u), Agg_j \leftarrow A_j(f_j(\overline{B}_j))] \bowtie_{v \in T_u} (R(v)))$ [LPS24]

Assume that as part of the initialization at node w, for each each tuple t_w, a frequency attribute $t_w.c$ has been initialized. Then in [LPS24], the additional attribute Agg_j is computed as follows (since AVG is computed by a combination of SUM and COUNT, it is omitted in the following descriptions):

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

If A_j is MIN or MAX, then we can simply apply these aggregate functions to the tuples of A_j and save the results in Agg_j. If, however, A_j is COUNT, then we have to distinguish between values that are null (which will not be counted) and non-null values. Since we have to be aware of how often a tuple appears, the Agg_j is set to the frequency of the respective tuple. In the case of A_j = SUM, we also have to take into consideration how often the value appears, so the sum of a certain attribute is the value times its number of appearance (the frequency) [LPS24].

For the propagation step, we also assume that the frequency attributes of the relations and tuples involved have already been computed. Furthermore, assume that the nodes u and u_1 are part of the path from w to r, and u_1 is the immediate child of u. Note that u_1 is the only child of u that is on the path from w to r, whereas $u_2, ...u_k$ are not. The following enumeration describes how Agg_j is passed from u_1 to u. For every tuple t of $R(u)$, the set $\{t_1, ..., t_n\}$ contains all tuples of $R(u_1)$ that are join partners for t. That means the tuples $\{t_1, ..., t_n\}$ contain values that need to be aggregated and propagated to t [LPS24].

- First suppose that $A_j \in \{MIN, MAX\}$. Then we set $t.Agg_j := A_j(\{t_1.Agg_j, ..., t_n.Agg_j\})$.
- Now let $A_j \in \{SUM, COUNT\}$. For every $i \in \{2, ..., k\}$, let s_i denote the sum of the frequencies of all join partners of t in $R(u_i)$. Then we set $t.Agg_j := (\sum_{m=1}^{n} t_m.Agg_j[u_1]) * \prod_{i=2}^{k} s_i$. Note that the value of $\sum_{m=1}^{n} t_m.Agg_j$ is aggregation of Agg_j over all join partners of t in child u_1 and $\prod_{i=2}^{k} s_i$ is the combined frequency values for each tuple gathered from the children $\{u_2, ...u_k\}$[LPS24].

We see that if $A_j \in \{MIN, MAX\}$, then the already aggregated values of the join partners $\{t_1.Agg_j, ..., t_n.Agg_j\}$ are aggregated once again into one value and saved in $t.Agg_j$. On the other hand, if $A_j \in \{SUM, COUNT\}$, then not only the aggregates of the tuples that can join with t have to be taken into account, but also the frequencies of values in children of u that are not on the path from w to r (that is, all nodes in $\{u_2,...,u_k\}$). This is why the already computed sums are are not just further aggregated, but also multiplied with the frequencies of their join partners from other children of u [LPS24].

Notably, even though the logical optimization now propagates more values, the number of tuples along the join tree is still the same as in regular semi-joins, as the relations are only extended by columns for attributes and frequencies, but never new tuples [LPS24]. That is, even though the size of the relations increases, it does so *horizontally* by adding attributes, but never *vertically* since no tuples are ever added. This means the evaluation of piecewise-guarded aggregate queries is still linearly bounded by the size of the data [LPS24].

Physical Optimization

The physical implementation of AggJoin combines Yannakakis-style evaluation and aggregation. A similar process has been described in [SOAK+19], albeit without computing and propagating frequency attributes of tuples. In this section, we will describe the join-less implementation of AggJoin by the authors of [LPS24].

As a preprocessing steps, a frequency attribute c is added to every relation and initialized as 1 for every tuple. Then, for every aggregate $A_j(f_j(\overline{B}_j))$ whose attributes are not guarded by the root, a node w is determined - the highest node in the join tree that

contains the attributes $\overline{B}_j$. Afterwards, every node along the path from w to root is extended by attribute Agg_j. The values of Agg_j are initialized as described above [LPS24].

After the pre-processing, AggJoin starts to propagate frequencies and aggregates. For the following explanations, the relations S and R will be the relations connected the nodes u_R and u_S in a join tree, where u_S is a child node of u_R. Frequencies are propagated the following way for a tuple $r \in R$ [LPS24]:

- $R \ltimes S$ must contain r
- find all possible join partners for r in S: $S' = S \ltimes \{r\}$
- calculate sc, i.e. the sum of the frequencies of all the tuples in S'.
- Set $r.c := r.c * sc$, that is the current frequency of tuple r is multiplied by the amount of possible join partners from S.

Before propagating aggregates, a small initialization step has to be performed for the aggregates SUM and COUNT that takes the frequency attributes of tuples in a node's subtree into account. Assume Agg_j has to be initialized in R. As already described in Section 4.5, for every tuple $r \in R$, Agg_j has to be multiplied by $r.c$. Note that $r.c$ is calculated as such: For every child u_i of u_R, the sum of frequencies of all the tuples in $R(u_i)$ that can join with t is calculated as sc_i. Then $r.c$ is multiplied with sc_i. [LPS24].

For the propagation of aggregates, two cases have to be distinguished: (1) S contains Agg_j and (2) S does not contain Agg_j. In case (1) the aggregates are processed the following way for every tuple $r \in R$ that has at least one join partner in S [LPS24]:

- If $A_j \in \{MIN, MAX\}$ then A_j is applied to the respective values of all possible join partners and the result is written to $r.Agg_j$. That means the minimal (or maximal) value of Agg_j of all join partners (and the current tuple) is propagated.
- If $A_j \in \{SUM, COUNT\}$, the sum of the aggregated values in the join partners are computed and then multiplied with $r.Agg_j$, which was initialized as the sum of frequencies of all tuples in S that join with R.

In case (2), the propagation involves less computation [LPS24].

- If $A_j \in \{MIN, MAX\}$, $r.Agg_j$ is left unchanged, as the value will be propagated from another child node.
- If $A_j \in \{SUM, COUNT\}$, then it is processed like a frequency, i.e. $r.Agg_j$ is simply multiplied by the sum of frequencies of S', the join partners in S for r.

```
SELECT MAX(m.price), o.name
FROM OWNER as o
NATURAL JOIN VISIT as v
NATURAL JOIN DOG as d
NATURAL JOIN MEDICATION as m
WHERE '2024-09-01' <= v.date and v.date < '2024-10-01'
GROUP BY o.name
```

Figure 4.3: An example query that is not fully guarded, but piecewise-guarded. The goal of the query is to find out which owner was willing to pay most for a single medication in September 2024.

Example of query evaluation with AggJoin. Looking at the query in Figure 4.3, we can see that the query is trivially group-guarded by relation OWNER, since only one attribute appears in the GROUP BY statement. However, this is not the same relation as the guard for the aggregate *MAX(m.price)*, which is in relation MEDICATION. The AggJoin operator can still resolve this query without materialization. As per definition of piecewise-guardedness (see Section 4.4), this query is piecewise-guarded: All elements of the GROUP BY statement are part of the same relation, and each individual aggregate consists only of attributes from one relation. Furthermore, the aggregate function(s) are part of {MIN, MAX, SUM, COUNT, AVG}. The corresponding join tree can be seen in Figure 4.4, relation OWNER (=O) is the root guard. The right side of the join tree can be processed by the standard procedure for guarded aggregate queries: for relevant tuples, a frequency attribute is added and propagated (as described in Section 4.3), semi-joins are applied. The left side of the join treeneeds to be processed a bit differently. Similar to [KAK+14], the aggregations needed in the final result are computed and propagated from the node containing the relevant attributes that is highest up in the join tree to the root. As described above, this propagation is realized by adding the respective columns Agg_j to each relation on the path to root. For our example in Figure 4.5, we see that *price* appears only in MEDICATION, but not in DOG or OWNER. That means we have to propagate the aggregate $MAX(m.price)$ along the path from MEDICATION to OWNER. In the first step, depicted in Figure 4.6, we can see that the frequency attributes have been initialized for all tuples. Agg_j has been initialized in relation that contain the attribute to be aggregated. Now, along with the frequencies, the column Agg_j that contains $MAX(m.price)$ of all possible join partners for DOG in MEDICATION will be added to DOG, which will eventually relay these values to OWNER. In the right join tree, frequency values are propagated from VISIT to OWNER. At the end of this propagation, the root guard will hold not only the attributes necessary for grouping but now after propagation also the computed aggregate values [LPS24]. The result of AggJoin, that is the result of propagating aggregates and frequencies while applying semi-joins, can be seen in Figure 4.7.

Implementation details. To sum up, AggJoin is a new physical operator, that does not

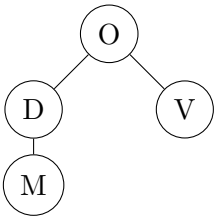


Figure 4.4: One possible join tree for the query in Figure 4.3, where the group guard OWNER has been chosen as root.

Owner

O_ID	FN	LN	Addr	Bank	Tel
101	A.	O.	Addr101	RB	Tel1
102	C.	O.	Addr102	VB	Tel2
103	E.	O.	Addr103	EB	Tel3
104	H.	O.	Addr104	OE	Tel4
105	S.	K.	Addr105	N26	Tel5

Dog

D_ID	Name	Breed	Weight	Medication	OwnerID
1	Bella	Labrador Retriever	25.0	Bravecto	101
2	Max	Beagle	10.0	NexGard	102
3	Luna	German Shepherd	30.0	Heartgard	103
4	Charlie	Golden Retriever	28.0	Bravecto	104
5	Daisy	Poodle	8.0	Bravecto	101
6	Rocky	Boxer	35.0	Revolution	101

Visit

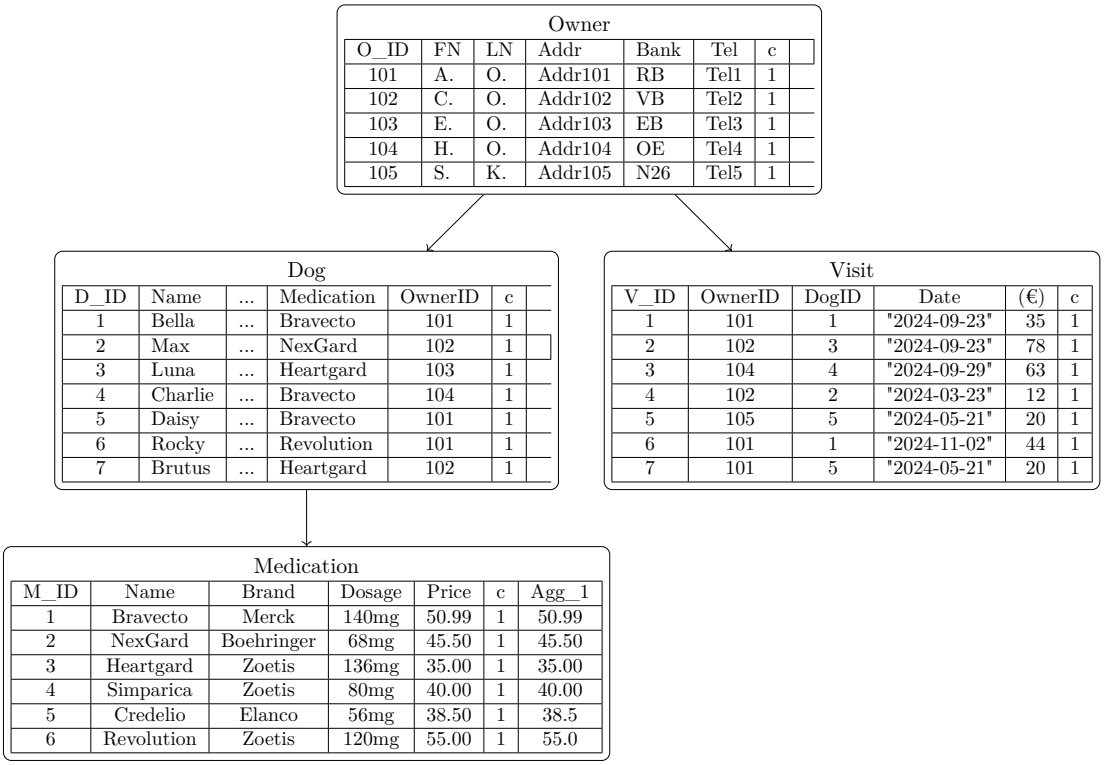
V_ID	OwnerID	DogID	Date	(€)
1	101	1	"2024-09-23"	35
2	103	3	"2024-09-23"	78
3	104	4	"2024-09-29"	63
4	102	2	"2024-03-23"	12
5	105	5	"2024-05-21"	20
6	101	1	"2024-11-02"	44
7	101	5	"2024-05-21"	20

Medication

M_ID	Name	Brand	Dosage	Price
1	Bravecto	Merck	140mg	50.99
2	NexGard	Boehringer	68mg	45.50
3	Heartgard	Zoetis	136mg	35.00
4	Simparica	Zoetis	80mg	40.00
5	Credelio	Elanco	56mg	38.50
6	Revolution	Zoetis	120mg	55.00

Figure 4.5: A more detailed join tree for Figure 4.4, containing the relations Owner O, Dog D, Visit V, and Medication M.

only keep track of frequencies, but also of aggregate values. This operator was integrated in Spark by the authors of [LPS24] with as minimal change as possible to the original processes in Spark. AggJoin was implemented in three variations: shuffled-hash join, sort-merge join and broadcast-hash join. Since these are the three join types that Spark naturally resorts to, the optimizations that Spark applies under the hood can still be applied before the AggJoin. In Algorithm 4.1 the implementation of AggJoin as Hash Join (with additional aggregate and frequency propagation), also called AggHashJoin, can be seen. Notably, the hash phase, which also includes partitioning, is not affected by the extension. Only the join phase is changed [LPS24]. To apply the AggHashJoin to two relation R and S to be joined (where R is the parent relation), the algorithm takes four lists as input: List R and S, that contain tuples that have join partners; List I_S



Owner

O_ID	FN	LN	Addr	Bank	Tel	c
101	A.	O.	Addr101	RB	Tel1	1
102	C.	O.	Addr102	VB	Tel2	1
103	E.	O.	Addr103	EB	Tel3	1
104	H.	O.	Addr104	OE	Tel4	1
105	S.	K.	Addr105	N26	Tel5	1

Dog

D_ID	Name	...	Medication	OwnerID	c
1	Bella	...	Bravecto	101	1
2	Max	...	NexGard	102	1
3	Luna	...	Heartgard	103	1
4	Charlie	...	Bravecto	104	1
5	Daisy	...	Bravecto	101	1
6	Rocky	...	Revolution	101	1
7	Brutus	...	Heartgard	102	1

Visit

V_ID	OwnerID	DogID	Date	(€)	c
1	101	1	"2024-09-23"	35	1
2	102	3	"2024-09-23"	78	1
3	104	4	"2024-09-29"	63	1
4	102	2	"2024-03-23"	12	1
5	105	5	"2024-05-21"	20	1
6	101	1	"2024-11-02"	44	1
7	101	5	"2024-05-21"	20	1

Medication

M_ID	Name	Brand	Dosage	Price	c	Agg_1
1	Bravecto	Merck	140mg	50.99	1	50.99
2	NexGard	Boehringer	68mg	45.50	1	45.50
3	Heartgard	Zoetis	136mg	35.00	1	35.00
4	Simparica	Zoetis	80mg	40.00	1	40.00
5	Credelio	Elanco	56mg	38.50	1	38.5
6	Revolution	Zoetis	120mg	55.00	1	55.0

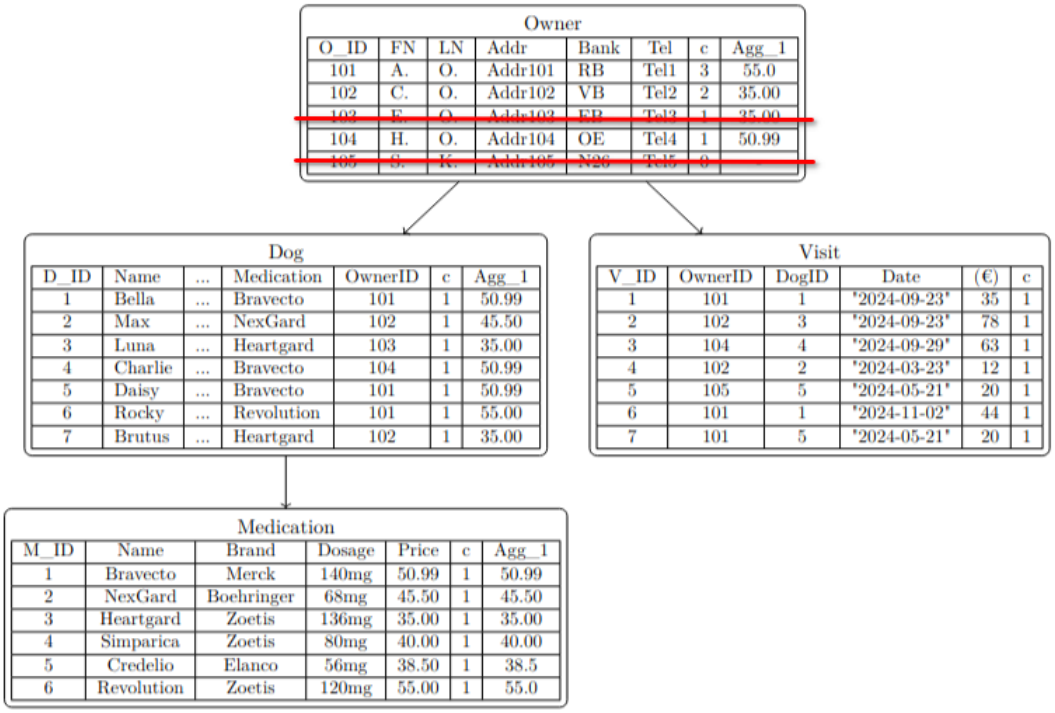
Figure 4.6: A join tree for a query that contains the relations Owner O, Dog D, Visit V, and Medication M, with added initialized columns c and Agg_1 for the respective relations.

containing aggregates that are present in both R and S; List I_R containing aggregates that are only present in R. In the function *AggHashJoin*, *sc* which will eventually hold the sum of frequencies of join partners of S for tuples in R, is initialized with 0. Then, every attribute in I_S is also initialized: SUM and COUNT are initially 0, whereas MIN is initialized as a maximum value (depending on the environment) and MAX as a minimal value, both environment variables are assumed to be stored in *init[s]*. The following foreach-loop over S aggregates the frequencies and values to be propagated from S, which are then used by the next for-loop over R to add the missing values from S, that is propagating frequencies and attributes to R. Finally, the frequencies have to be multiplied for SUM and COUNT before the list of tuples is returned [LPS24].

Algorithm 4.1: Hash Join with aggregate propagation

Input: Two lists R, S of tuples with the same values of the join attributes;
Input: List $I_S = \{s_1, \ldots, s_m\}$ of indices of aggregate attributes Agg_{s_i}, present in both S and R;
Input: 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          else 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(vals, t.Agg_s);
11             else if A_s ∈ {SUM, COUNT} then vals ← 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             else if A_s ∈ {SUM, COUNT} then t.Agg_s ← t.Agg_s · vals;
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         return t;
24     end
```



Owner

O_ID	FN	LN	Addr	Bank	Tel	c	Agg_1
101	A.	O.	Addr101	RB	Tel1	3	55.0
102	C.	O.	Addr102	VB	Tel2	2	35.00
103	E.	O.	Addr103	EB	Tel3	1	35.00
104	H.	O.	Addr104	OE	Tel4	1	50.99
105	S.	K.	Addr105	N26	Tel5	0	

Dog

D_ID	Name	...	Medication	OwnerID	c	Agg_1
1	Bella	...	Bravecto	101	1	50.99
2	Max	...	NexGard	102	1	45.50
3	Luna	...	Heartgard	103	1	35.00
4	Charlie	...	Bravecto	104	1	50.99
5	Daisy	...	Bravecto	101	1	50.99
6	Rocky	...	Revolution	101	1	55.00
7	Brutus	...	Heartgard	102	1	35.00

Visit

V_ID	OwnerID	DogID	Date	(€)	c
1	101	1	'2024-09-23'	35	1
2	102	3	'2024-09-23'	78	1
3	104	4	'2024-09-29'	63	1
4	102	2	'2024-03-23'	12	1
5	105	5	'2024-05-21'	20	1
6	101	1	'2024-11-02'	44	1
7	101	5	'2024-05-21'	20	1

Medication

M_ID	Name	Brand	Dosage	Price	c	Agg_1
1	Bravecto	Merck	140mg	50.99	1	50.99
2	NexGard	Boehringer	68mg	45.50	1	45.50
3	Heartgard	Zoetis	136mg	35.00	1	35.00
4	Simparica	Zoetis	80mg	40.00	1	40.00
5	Credelio	Elanco	56mg	38.50	1	38.5
6	Revolution	Zoetis	120mg	55.00	1	55.0

Figure 4.7: The result of applying AggJoin to solve the query in Figure 4.3. The columns c and Agg_1 have been added and computed for every tuple.

CHAPTER 5

Spark and SparkSQL

Spark [Apa09] is a distributed data processing/ computing framework that was developed in 2009/2010 at Berkeley's AMPLab and donated to Apache in 2014 as an answer to the ever rising volume of data to be processed. Its original purpose was to serve as an abstraction of MapReduce (see Section 5.1), but since the SparkSQL module was added, it has also become a popular distributed SQL query engine [Pip21, LPS24, GPS22, AXL+15].

5.1 Spark

As written above, Spark is a distributed data processing framework. It offers APIs for Java, Python and Scala and integrates features for streaming, machine learning, graph processing and querying relational data [AXL+15, Apa09]. Its main building blocks are so-called resilient distributed datasets (RDDs). An RDD is a set of tuples or data elements (the partitions) which belong together semantically (e.g. tuples of a relation), but are distributed over several nodes of the Spark cluster. The "resilient" part of the name refers to a property that is close to the functional paradigm, that is writing functions and procedures without mutating either the data or the state. In Spark, an operation on a batch of data does not modify the original data, but instead creates its own copy to work on [GPS22, Pip21]. When RDDs were introduced, the authors described them as "fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators." [ZCD+12]. RDDs are fault-tolerant, because they only allow coarse-grained action and transformations like map, filter or join. These actions are then repeatedly applied to different batches of data and logged, which creates the so-called *lineage* of a dataset. Should a partition get lost for whatever reason, it can be recomputed by applying the actions in the lineage to the original RDD [ZCD+12]. This lineage can also be described as "logical plan" to compute a dataset, but it is evaluated

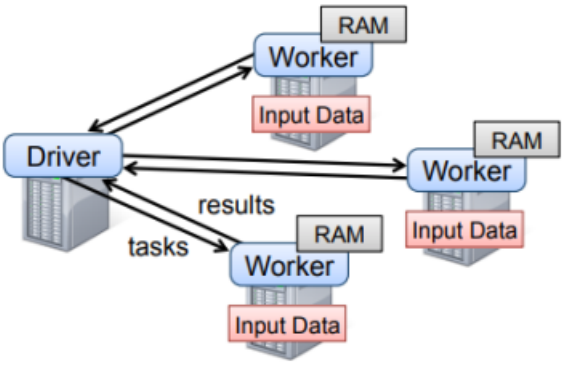


Figure 5.1: Spark Architecture [ZCD+12]

lazily, i.e. unless for example an output operation necessitates the actual computation of a value, the computation is postponed [AXL+15].

But what exactly are the nodes of a Spark cluster that hold the partitions? Spark is based on a driver-worker architecture (see Figure 5.1). The driver program connects to the cluster of workers, and defines the operations to be invoked and simultaneously keeps track of each partition's lineage. Both workers and the driver have their own RAM space and CPU core(s). Once a worker has finished its task, it reports back to the driver. Since the driver holds all the information about lineages, a crashed node can be replaced by a new one and resume its work after invoking the transformations of the former lineage. If the driver crashes, however, the application shuts down, too [ZCD+12, Pip21].

The classic and original purpose of Spark was to provide an abstraction over MapReduce, a programming model developed in 2008 by Google engineers [DG08]. It itself is an abstraction to aid in the computation of derived data from huge volumes of raw data in a parallel, distributed and safe manner. Simply put, the model takes a set of input key/value pairs, and produces a set of output key/value pairs or just values. More precisely, the mapping step iterates over the key/value pairs and groups the values in accordance to a new, intermediate key. The reduce function then recognizes the new keys and reduces all values with the same key to a final output. The task of the user is to implement the map function and the reduce function according to the task at hand (see Figure 5.2). As an example, imagine a MapReduce function that counts the occurrence of words in a text. The map function accepts the lines of the text, so each line number is a key and each line a value. To count the words, each word becomes an intermediate key with the value 1. The reducer now adds all values that have the same key - the word count is done [DG08]).

Spark was introduced because even though MapReduce became a popular model, it required the knowledge of either C++ or Java. Spark offered a layer of abstraction that allowed users to create and manipulate RDDs instead of writing Java or C++ code [Pip21].

map	(k1,v1)	→ list(k2,v2)
reduce	(k2,list(v2))	→ list(v2)

Figure 5.2: The core concepts of MapReduce: The mapping step sorts the values according to intermediate keys (here: k2) and reduce then produces the final output, that is relating every value v2 that has the same key k2 [DG08].

5.2 SparkSQL

SparkSQL is the SQL module of Spark, i.e. the SQL framework within the Spark query engine. Its main focus is t be able to process data using "a combination of both relational queries and complex procedural algorithms" [AXL+15]. This is achieved by two components: The DataFrame API and the Catalyst Optimizer.

5.2.1 The Dataframe API

Similar to the dataframe concept of the R language [R-p93], SparkSQL's DataFrame API offers support for external data sources and Spark's native RDD for relational operations. A dataframe is similar to a table, but can be manipulated in a straightforward way like RDDs (see Fig 5.3). However, other than RDDs, dataframes do hold schema information of the data they contain and are generally untyped. If one wants to introduce types (e.g. assign a type to each row in a relation), one can use the DataSet API [1] of Spark [Apa09, AXL+15].

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

Figure 5.3: The two dataframes *employees* and *depts* are joined and filtered with additional grouping and aggregating [AXL+15].

5.2.2 The Catalyst Optimizer

SparkSQL furthermore contains the Catalyst optimizer, that applies rule-based and (limited) cost-based optimization to each query in the multi-step query planning phase: analysis, logical optimization, physical planning, and code generation (see Figure 5.4).

[1] https://spark.apache.org/docs/3.5.1/api/java/org/apache/spark/sql/Dataset.html

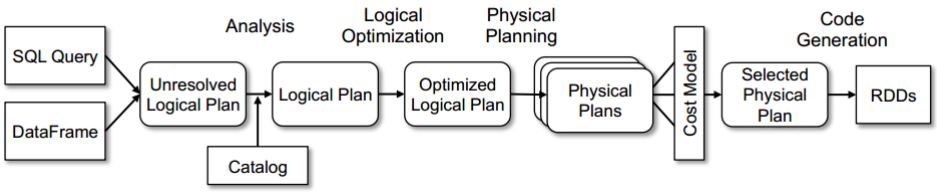


Figure 5.4: Each step of query processing and execution planning in SparkSQL. Rectangles with rounded corners represent query trees. [AXL+15]

The process accepts two kinds of inputs: Either a dataframe object (and its included lineage) or an abstract syntax tree (AST) constructed by a SQL query parser. Either way, in the first step the received set of instructions is transformed into an *unresolved logical plan*. It is called unresolved, because in this step the attributes are neither type-checked nor linked back to the relations they stem from. In the next step, these uncertainties are resolved with the help of the catalog, an object that tracks the tables of a schema. As a result, relations are looked up, attributes mapped and types determined. We obtain a *logical plan*. A logical plan can be imagined as a further refinement of a join tree. Other than a join tree, a logical plan is a binary tree and holds information about processing steps like filtering, projecting and joining. From this point on, optimizations take place. The *optimized logical plan* is the result of rule-based optimization like folding of constants (computing constants at compile time rather than runtime), predicate pushdown, projection pruning (unnecessary columns are removed from the query processing pipeline), null propagation, Boolean expression simplification etc. [AXL+15]. With this optimized logical plan as a basis, SparkSQL (or rather Catalyst) now creates one or more physical plans by applying cost-based optimization. Notably, SparkSQL offers only limited cost-based rules that only select a certain join algorithm based on statistical values (if available). Additionally physical optimizations regarding pipeline processing, i.e. further projections and filtering into a *map* expression, are applied [AXL+15].

In the last step, the *code generation stage*, Java bytecode is generated. To avoid the complexities of typical code generation engines that almost amount to stand-alone compilers, SparkSQL takes advantage of the Scala language, more precisely the "quasiquote" library [SBO13]. "Quasiquote" allows the construction of ASTs that can then be fed to the Scala compiler to generate bytecode at runtime, while still allowing the compiler to add more optimizations in case they were missed in the steps before [AXL+15].

5.2.3 Static Optimizations of SparkSQL Catalyst

As opposed to other modern DBMS like PostgreSQL [2] or Oracle [3], the Catalyst optimizer does not have advanced cardinality estimation. It only includes basic cardinality

[2] https://www.postgresql.org/docs/current/row-estimation-examples.html
[3] https://blogs.oracle.com/optimizer/post/cardinality-and-dynamic-statistics

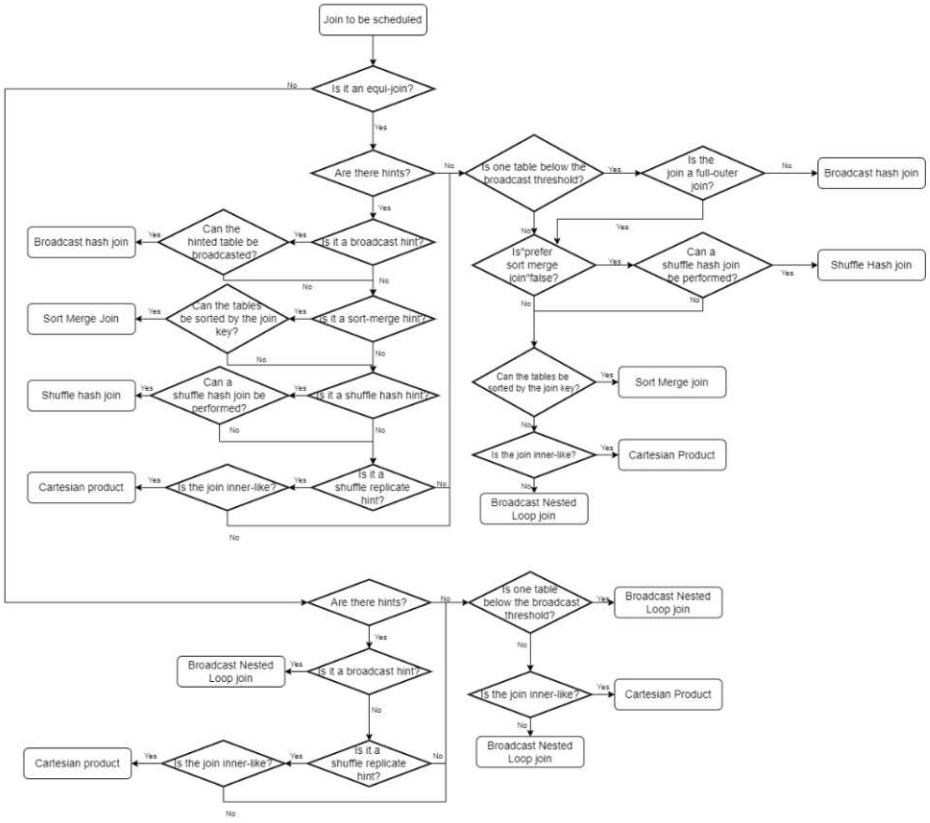


Figure 5.5: Flowchart on join type selection in Spark [Pip21]

estimation via statistics collection (if available), but it lacks advanced histograms (e.g. Equi-depth histograms [MD88, AC99, CHW+22]) and multi-column correlation tracking, which often leads to inaccurate estimations [AXL+15]. However, SparkSQL does come with other in-built optimization techniques, like filter push-down, complex join-type selection and lazy evaluation [AXL+15, pro, Pip21]. As an example for the delicate complexity of the Catalyst optimizer, the reader is invited to take a look at Figure 5.5, that gives an overview of how join type selection is done in Spark.

5.2.4 Dynamic Optimization: Adaptive Query Execution.

Since the introduction of Spark 3.3.0., adaptive query execution (AQE) is an available feature in Spark. AQE utilizes runtime statistics to select a physical query plan or change it, should another one become the most efficient query execution plan during the execution (i.e. due to circumstances not foreseen by Spark's limited cardinality estimation). It enables four new optimizations during runtime: Automatically coalescing post-shuffle partitions, converting Sort Merge Joins into Broadcast Hash Joins, converting Sort Merge Joins into Shuffle Hash Joins, optimizing skewed joins [Pip21, pro]. As can be seen in Figure 5.6, AQE is a communication between the execution stage and the

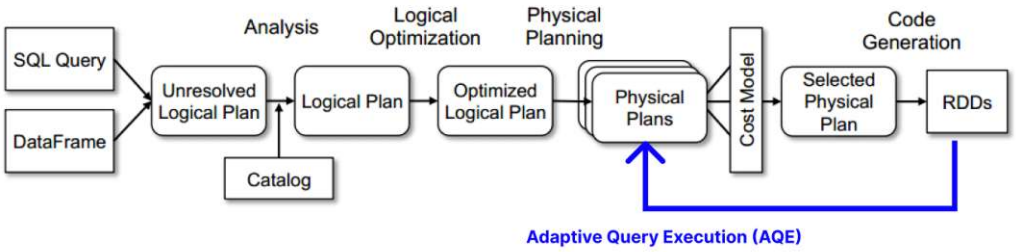


Figure 5.6: Adaptive Query Planning (AQE) in Spark. Information gained during execution is used to switch to a different execution plan [AXL+15, Pip21]

physical planning stage.

CHAPTER 6

Implementation and Evaluation

6.1 Benchmark Datasets and Analysis

6.1.1 Benchmarks

The algorithm developed in [LPS24] was benchmarked on five different datasets: STATS-CEB [HWW+21], SNAP [LK14], JOB [LGM+15], LSQB [MLK+21] and TPC-H [TPCb]. In this section, we will describe and analyze these benchmark datasets to find out whether they contain enough non-guarded queries to re-use them for our evaluations. Additionally, we will present Syn-TPC-H, a synthetically created dataset that focusses on non-guarded queries.

JOB. The JOB dataset stands for Join Order Benchmark and was introduced in 2015 in [LGM+15] in order to evaluate the quality of industrial-strength query optimizers. It consist of mulit-join queries that are based on the IMDB dataset[1], which contains real-world information about movies, actors, directors, etc. The goal was to create realistic queries in the form of Select-Project-Join (SPJ) blocks. The main challenge that these queries pose is the significant number of joins: each query has between 3 and 16 joins with an average of 8 joins per example.

SNAP. SNAP (Stanford Network Analysis Project [LK14]) is a dataset that focuses on evaluating graph queries (e.g. in [HW23]) and generally analyzing big graph structures like social networks. In [LPS24], the following graphs where used for evaluation:

Graph	Nodes	Edges	(un)directed
wiki-topcats	1,791,489	28,511,807	directed
web-Google	875,713	5,105,039	directed
com-DBLP	317,080	1,049,866	undirected

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

In the evaluation task, path queries with 3 to 8 joins (i.e. 4 to 9 edges on a graph) and three small tree-queries were taken into account.

STATS-CEB. The STATS / STATS-CEB [HWW+21] benchmark was created in 2021 in order to benchmark cardinality estimation (CardEst) methods and is similar to JOB, but contains some more advanced joins. The data that is queried is anonymized textual content from Stack Exchange.

LSQB. LSQB stands for "Large-Scale Subgraph Query Benchmark" [MLK+21] and as the name suggests, it serves as a benchmark for processing graph queries and the respective optimizers. Similar to SNAP, it is based on a social network like structure that can be scaled up and down. In [LPS24], a scale factor of 300 was used.

TPC-H The TPC-H dataset [TPCb] consists of business oriented queries that can be used to benchmark relational database systems. According to [TPCb], the queries aim to answer realistic, "critical business questions" and consists of 22 queries and 8 base relations [TPCb]
In addition to TPC-H, **TPC-DS** [TPCa] is based on similar data but contains more complex queries including sub queries, complex joins, nested expressions, etc. It is based on 24 relations and holds 99 queries [TPCa].

In the first version of [LPS24], the authors were able to apply their optimization to guarded queries and were able to process all queries in the STATS-CEB dataset [HWW+21] and the SNAP dataset [LK14], but only 5 out of 33 queries in the JOB dataset [LGM+15] were eligible for optimization. In LSQB [MLK+21] only 2 out of 9 could be processed and only 2 out of 22 in TPC-H [TPCb]. The newest version of the optimization proposed in [LPS24], AggJoin, was applicable to all queries in the STATS-CEB dataset [HWW+21] and the SNAP dataset [LK14]. Also, all 113 JOB queries could be processed by the new optimization. However, only 7 out of 22 TPC-H queries, 30 out of 99 TPC-DS queries and 2 out of 9 LSQB queries could be processed with AggJoin.

6.1.2 Analysis of the Benchmark Data

In this section, we will present an analysis that shows the internal structure of the above presented benchmark sets. All queries can be found in the GitHub repository of the DBAI Institute of TU Wien [2]. For our work, it is relevant to look at the complexity of queries, since very simple queries (see Section 4.3) are trivially guarded. In the analysis, we took each query and counted the relations involved, number of aggregates and number of attributes appearing in GROUP BY statements. Afterwards, these numbers were averaged for each benchmark as a measure of benchmark complexity. Furthermore, each query was checked in terms of guardedness, both in terms of aggregate-guardedness and group-guardedness. The sqlglot library was used to carry out the query parsing [3]. Please

[2] https://github.com/dbai-tuw/spark-eval/tree/main/benchmark
[3] https://github.com/tobymao/sqlglot

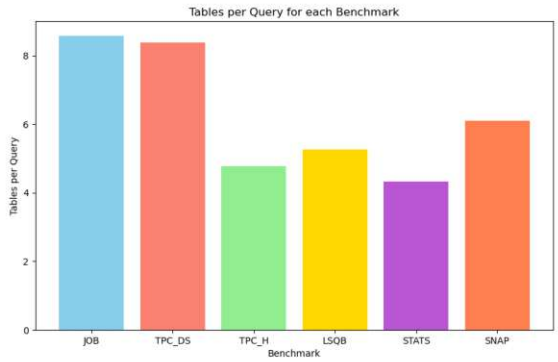


Figure 6.1: The average amount of relations that are involved in the queries of the Benchmarks JOB [LGM+15], TPC-DS [TPCa], TPC-H [TPCb], LSQB [MLK+21], STATS [HWW+21], and SNAP [LK14].

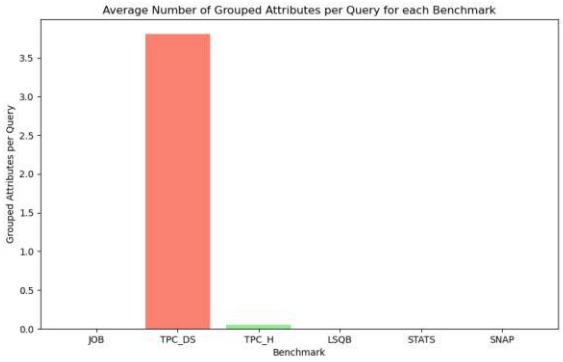


Figure 6.2: The average amount of attributes that appear in GROUP BY statements in the Benchmarks JOB [LGM+15], TPC-DS [TPCa], TPC-H [TPCb], LSQB [MLK+21], STATS [HWW+21], and SNAP [LK14].

note that due to some technical limitation regarding specific SQL dialects, some queries had to be skipped. Also note that since we are interested in the number of attributes appearing as aggregated or grouped attributes, expressions like COUNT(*) that appear quite often in SNAP or STATS, are counted as 0 aggregation attributes since no concrete attribute appears. The results of this analysis are presented in Figure 6.1, Figure 6.2 and Figure 6.3. An additional analysis of the benchmarks that also includes information about acyclicity and whether the query contains only equi-joins can be seen in Figure 6.4, results taken from [LPS24].

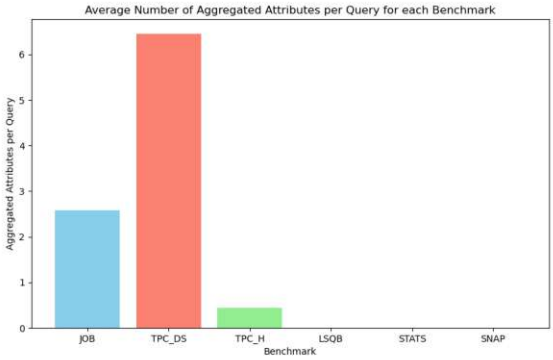


Figure 6.3: The average amount of attributes that appear in aggregates in the Benchmarks JOB [LGM+15], TPC-DS [TPCa], TPC-H [TPCb], LSQB [MLK+21], STATS [HWW+21], and SNAP [LK14].

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

Figure 6.4: Overview of the applicability of the AggJoin developed in [LPS24]s. The table shows the number of queries (#), equi-join aggregate queries (⋈-agg), acyclic queries (acyc), piecewise-guarded queries (pwg), guarded queries (g), and 0MA queries [LPS24].

6.1.3 Additional Synthetic Data

As one can see in Figures 6.1, 6.2 and 6.3, even though all benchmark datasets contain queries that involve more than four relations on average, the average number of attributes that appears in aggregates or GROUP BY statements is fairly small and most of the analyzed queries are already guarded or piecewise-guarded. This means while the mentioned benchmarks worked well to showcase the original algorithm presented in [LPS24], their lack of non-guarded queries makes them unsuitable as benchmark for our implementation. For this reason, a new synthetic dataset of 45 queries that correspond to the TPC-H schema was created. Of these 45 queries 35 are non-guarded and 10 are (piecewise-)guarded in order to ensure that PartAggJoin does not interfere with AggJoin in case of (piecewise-)guarded queries. We will call the schema Syn-TPC-H.

The synthetic queries vary in complexity: Some involve only 3 relations and require only one join with full materialization to become guarded queries, others involve up to 6 relations and require multiple joins in order to be transformed into a guarded query. An exact overview of the characteristics of the whole Syn-TPC-H dataset is given in the

table depicted in Table A.1 in the Appendix.

For the non-guarded queries in the dataset, which will be of special focus in this work, the following characteristics were extracted:

- Mean number of tables per query: 3.41
- Minimal number of tables per query: 2
- Maximal number of tables per query: 6
- Mean number of grouped attributes per query: 2.18
- Minimal number of grouped attributes per query: 2
- Maximal number of grouped attributes per query: 4
- Mean number of aggregated attributes per query: 1.68
- Minimal number of aggregated attributes per query: 1
- Maximal number of aggregated attributes per query: 4

6.2 Enforcing Guardedness

As described in Section 4.5, the AggJoin optimization only works on guarded or piecewise-guarded queries. That means that queries that contain multiple attributes in the GROUP BY clause that do not stem from the same relation cannot profit from the Yannakakis-style optimization provided by AggJoin.

For example the query in Figure 6.5 cannot be evaluated by AggJoin since the grouping attributes stem from different relations, so it is impossible that the root node of a join tree contains both attributes (here *o_orderpriority* and *l_shipmode*) initially.

In this section, we will discuss three different methods that can be used to create guardedness for previously non-guarded queries. The *Upfront Joins* method identifies relations that contain the non-guarded attributes and joins them, thus creating a new intermediate relation that then serves as a guard to the query. The second method, *Partial AggJoin (PartAggJoin)* works similarly, but does not join tables before evaluating the query, but during evaluation. The third method is mentioned in this work for reasons of completeness. It is currently being developed by the authors of [LPS24]. This method that is called *GroupAggJoin* applies the same principle as AggJoin to attributes in the GROUP BY clause: attributes that are non-guarded are propagated to the top of the join tree, which transforms the root-relation of the tree to a query guard. In the following sections all three approaches will be described in detail.

```
SELECT
o_orderpriority AS Order_Priority,
l_shipmode AS Ship_Mode,
SUM(l_extendedprice) AS Total_Revenue
FROM orders
JOIN
lineitem
ON orders.o_orderkey = lineitem.l_orderkey
GROUP BY
o_orderpriority,
l_shipmode
```

Figure 6.5: A query over the TPC-H database [TPCb] that is non-guarded (o_orderpriority and l_shipmode are from different relations) and cannot be evaluated by AggJoin.

6.2.1 Upfront Joins

The method presented in this section is called *Upfront Joins* because it identifies and joins relations needed for a guard before actually processing and evaluating the query.

The process consists of the following steps:

1. **Categorizing the query.** In the first step, the query needs to be parsed in order to determine which category it belongs to: guarded, piecewise-guarded or non-guarded.

   In the same step, the algorithm identifies which relations can form the root guard by relying on schema information. This means the different relations that contain attributes of the GROUP BY clause need to be determined.

2. **Traverse the join tree.** In this step, the goal is to find a path that contains all the identified relations from step 1. For this, the join tree is traversed until a path (or subtree) containing all relations is found.

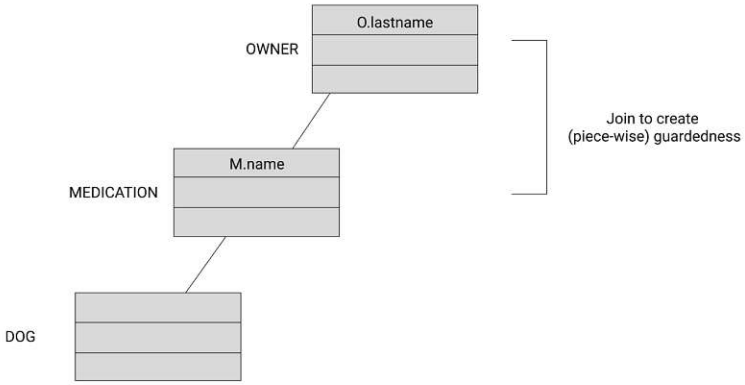
3. **Create a guard.** A guard can now be created by joining the identified relations along the path (or subtree) calculated in step 2. The resulting intermediate relation now serves as a guard to the query at hand.

4. **Process the query with AggJoin.** Now that a group guard was created, processing the query can be continued by utilizing the optimization provided by AggJoin.

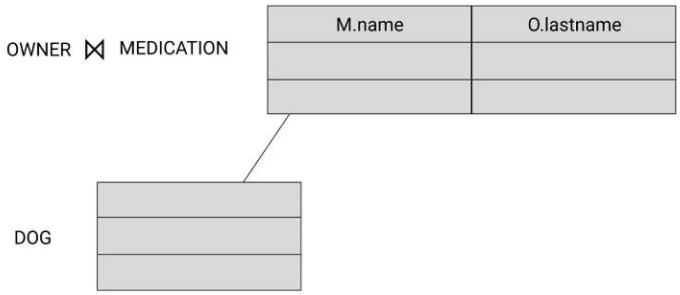
   As an example, the process of enforcing guardedness for the query shown in Figure 6.6 is presented below in Figure 6.7. In Figure 6.7a, one can see the corresponding join tree, consisting of three nodes. In order to create a guard, the

```
SELECT M.name, O.lastname, SUM(M.price)
FROM DOG as D
NATURAL JOIN MEDICATION as M
NATURAL JOIN OWNER as O
GROUP BY M.name, O.lastname
```

Figure 6.6: An example of a non-guarded query (M.name and O.lastname are from different relations)



(a) Join Tree of the query in Figure 6.6.



(b) Join tree after creating a guard by joining two relations.

Figure 6.7: Enforcing guardedness via upfront joins.

relations OWNER and MEDICATION have to be joined, since the GROUP BY clause contains attributes from both of these relations. In Figure 6.7b, the jointree has a new node resulting from joining the two relations. In this form, the query is guarded and can be further processed by AggJoin.

6.2.2 Partial AggJoin (PartAggJoin)

Other than the *Upfront Joins* method, *PartAggJoin* enables integrating the necessary minimal materialization directly into the AggJoin procedure. To this end, the algorithm selects the appropriate logical join type at each node of the logical plan of *a non-guarded query*. Note that the query type is known in advance, since AggJoin automatically deduces it during execution. The core functionality of *PartAggJoin* consists of the following case distinction. Assume that at node u, the relations R and S have to be joined.

1. Case: Neither of the relations R and S contains attributes that are part of the GROUP BY statement. In this case, an AggJoin is performed.

2. Case: At least one of the relations R or S contains attributes that are part of the GROUP BY statement. In this case a join with full materialization has to be selected, since the grouping attributes have to be propagated for each tuple.

To this end, it is necessary to hook into each step of the traversal of the logical plan in the logical layer and extend the given code with the functionality that follows. Just as AggJoin (see Section 4.5), PartAggJoin decides which aggregates have to be initialized/propagated and also takes care of initializing/propagating frequencies at those nodes of the join tree that cannot apply AggJoin. To find out when (not) to apply AggJoin, *PartAggJoin* also needs to check whether the current node has so-called *applicable grouping attributes*. If the current node does contain these grouping attributes, i.e. one of the relations to be joined contains one or more grouping attributes, the logical plan has to be modified so that the system does not not apply the AggJoin operator, but instead uses a "normal" join operator with materialization. If none of the relations contains grouping attributes, AggJoin is selected as logical join operator.

Eventually, this results in a join tree whose subtrees will be evaluated by different methods: Subtrees without grouping attributes will be processed by AggJoin without any materialization, whereas relations of subtrees that do contain grouping attributes will be joined by ordinary join methods (plus the additional propagation of pass-through values for frequency and aggregation-values).

As an example, we take a look at the query depicted in Figure 6.8. It is non-guarded, as the two attributes in the GROUP BY clause are not contained in the same relation. In Figure 6.9, we can see a possible join tree for this query and the result of applying PartAggJoin to evaluate the query. Since relation C and A contain grouping attributes, every relation along the path from C to A has to be joined by a "normal join". Subtrees that do not contain grouping attributes can be fully evaluated by AggJoin. Note that this example will be revisited in Section 6.3 and evaluated in a more detailed step-by-step approach.

```
SELECT A.a, C.x, SUM(z)
FROM A
NATURAL JOIN B
NATURAL JOIN C
NATURAL JOIN D
...
GROUP BY A.a, C.x
```

Figure 6.8: A non-guarded query. Note that "..." is a placeholder for one or more further joins that involve relations that hold no grouping attributes.

Figure 6.9: One possible join tree for the query in Figure 6.8. The logical join operators (written in red) are selected by PartAggJoin accordingly.

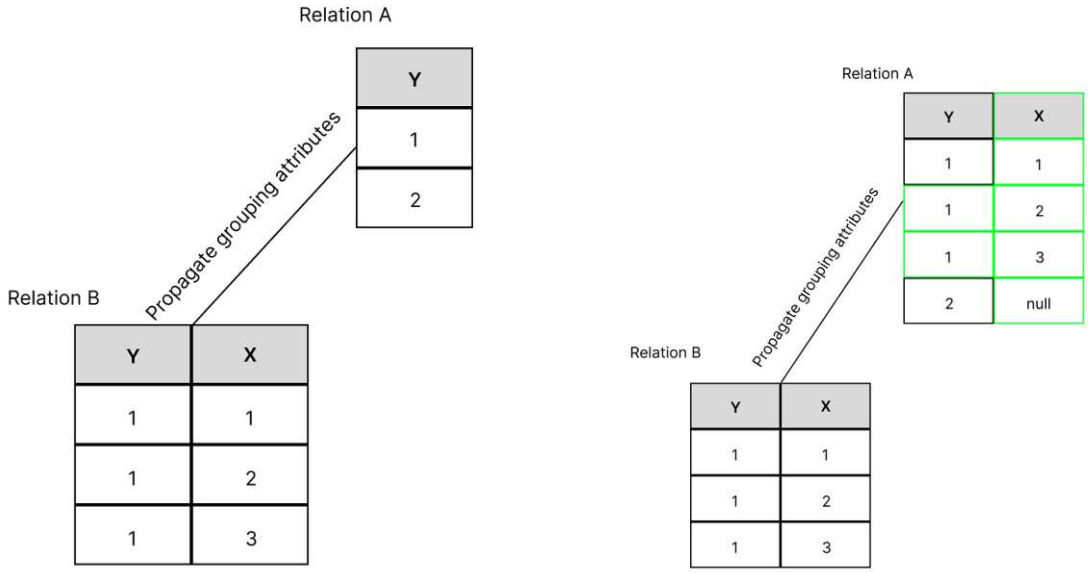
6.2.3 GroupAggJoin [LPS25]

An alternative to the *Upfront Joins* method and the *PartAggJoin* is *GroupAggJoin*. Since it is an extension for AggJoin (see Section 4.5), it works very similarly. However, whereas AggJoin propagates aggregates up the join tree, GroupAggJoin has to propagate attribute values. The method is currently developed by the authors of [LPS24] and thus mentioned here for reasons of completeness.

GroupAggJoin enforces guardedness by not only propagating the frequency of tuples up the join tree, but also by propagating real *attribute values*. The reason for this is that in the case of piecewise-guarded queries, not all aggregated attributes have to be present in the guard initially, but when the aggregation is performed at the end of the query processing, all aggregated attributes have to be present in the root guard, otherwise one cannot evaluate the query without materialization. This way of enforcing guardedness can also be applied to attributes that appear in the GROUP BY statement. If two or more attributes from the GROUP BY statement stem from different relations, GroupAggJoin propagates them up the join tree alongside frequencies and needed aggregates. The extension can easily be added to the existing implementation. Assume a grouping attribute of the form $\gamma(f_j(g_1, ..., g_n))$, where γ denotes the grouping operator, f_j an arbitrary function and $g_1, ..., g_n$ the attributes used for grouping. One example would be *GROUP BY LOWER(c.name, a.year)*. Propagating grouping attributes is even simpler than propagating aggregated attributes: Since the grouping is done by the values (or respectively by $f_j(g_1, ..., g_n)$), there is no special initialization value. If a grouping attribute g_i is not present in the root node u_r, we identify node u_w as closest descendant node that contains g_i and add g_i to every node on the path from u_w to u_r. Then, the join can proceed.

Recalling Algorithm 4.1, we can see that AggHashJoin has four inputs: the lists R and S that contain the tuples of relation R and S that have join partners; a list I_S of aggregate attributes present in R and S and a list I_R that contains aggregates of R that are not present in S. For GroupAggJoin (or, more precisely GroupAggHashJoin), one input parameter has to be added: a list G_S that contains the grouping attributes present in the query that is currently processed. Then, a map is created using the grouping attributes as keys and adding the aggregations calculated as the loops progress. The output type is now slightly different: the algorithm returns a map with the grouping attributes as a (compound) key and the corresponding tuples of R, extended with frequencies and attributes, as values.

In terms of memory and computational cost, this new operator does add new tuples. Even though no full join is materialized, the intermediate relations can increase sharply in size if the data is skewed, as can be seen in Figure 6.10a and 6.10b. However in cases where no attribute propagation is needed, the procedure is just a basic semi-join [LPS24].



(a) Before applying GroupAggJoin

(b) After applying GroupAggJoin

Figure 6.10: Comparison of the join tree before and after applying GroupAggJoin. In this example, the query contains *GROUP BY B.X*, hence all values of X have to be propagated from relation B to relation A.

6.2.4 Possible Pitfalls and Consequences

In this section, we will look at possible problems that are likely to occur in a concrete practical situation, i.e. when implementing *Upfront Joins* or *PartAggJoin* in a Spark/SparkSQL environment. While this section will start out with a more general problem that regards all of the above methods, we will then focus on comparing problems between *Upfront Joins* and *PartAggJoin*, since *GroupAggJoin* is implemented by other researchers [LPS25].

Cardinality estimation in Spark/SparkSQL. This problem regards all of the methods described above. The Catalyst optimizer in SparkSQL offers various optimizations (see Chapter 5), but has limited cardinality estimation. Even though SparkSQL's Catalyst Optimizer does include basic cardinality estimation via limited statistics collection, it lacks advanced methods, which makes its estimates less precise than traditional database systems like PostgreSQL [4] or Oracle [5] [AXL+15]. Consequently, if joins are executed during query processing (e.g. during *Upfront Joins* or *PartAggJoin*), one can only roughly guess how expensive an individual join is, let alone two or three. But even if no full join is materialized, e.g. during *GroupAggJoin*, this is problematic. *GroupAggJoin* also adds new rows to relations and consequently the cardinalities of the involved relations play a

[4] https://www.postgresql.org/docs/current/row-estimation-examples.html
[5] https://blogs.oracle.com/optimizer/post/cardinality-and-dynamic-statistics

more critical role than - for example - in the case of AggJoin: Since AggJoin operates on piecewise-guarded queries, one can always expect a performance improvement, since materialization is completely avoided. This is not the case for the methods described above. Conclusively, the lack of sophisticated cardinality estimation in Spark/SparkSQL is a drawback for *Upfront Joins*, *PartAggJoin* and *GroupAggJoin*, because without further statistics to rely on, there is no way to estimate how big the intermediate result will be.

This problem might be mitigated by applying statistical feature extraction and machine learning in order to identify key factors that might give insight to hidden costs. This possibility will be discussed in the outlook given in Chapter 8.

The following paragraph will now focus more on the comparison of *Upfront Joins* vs. *PartAggJoin* that were encountered during the implementation of a Proof-of-Concept for both methods.

Implementation Problems: *PartAggJoin* vs *Upfront Joins*. As we recall, the *Upfront Joins* method consists of five steps:

1. Categorize the query: guarded, piecewise-guarded or non-guarded
2. Identify the relations that can form a root guard if joined.
3. Traverse the join tree to find a path or subtree that contains all the relations identified in step 2.
4. Create a guard by joining these relations.
5. Feed back the newly created join tree to AggJoin.

PartAggJoin, on the other hand, does not involve any pre-processing of the logical plan. It is directly integrated in the AggJoin procedure and selects an appropriate join type for every node in the logical plan:

1. Case: Neither of the relations to be joined contain attributes that are contained in the GROUP BY statement. In this case, an AggJoin is performed.
2. Case: At least one of the relations to be joined contains attributes that are part of the GROUP BY statement. In this case a join with full materialization has to be selected, since the grouping attributes have to be propagated for each tuple.

When it comes to the actual implementation, the straightforward approach of *Upfront Joins* proved to have some pitfalls. We will discuss the two most pressing problems: the difficulty to feed back the newly created join tree to AggJoin and the overhead created by redundant traversals.

Back-propagating the newly created join tree. One major hindrance in implementing the *Upfront Joins* method turned out to be the separation of the different layers

in SparkSQL and communicating in both directions. Once a logical plan is created, it undergoes certain optimizations. Based on the optimized logical plan, one or more physical plans are generated, and the most promising one (according to a cost model) is realized. Applying the *Upfront Joins* method would involve forcing both logical and physical plan to include joins, then realizing these joins and then relaying this newly created join tree to the logical layer. One problem was that materializing joins does not immediately create a join tree - it mainly restructures data by creating an intermediate table (the guard), that has to be then integrated with the original query and a new logical plan. But to integrate the guard, there first needs to be a channel to propagate it back from the physical to the logical layer of the query engine, which is as of now - to the best of our knowledge - not intended by the engine. Even Spark's own Adaptive Query Execution (AQE), which does use a channel that allows some back propagation does not allow direct communication between the physical and logical layer. So in order to propagate the intermediate results back, we would have needed to either try to tweak and enhance AQE channels for our own purpose or create a new side channel, similar to [BKN24]. Knowing the complexity of the SparkSQL query engine, neither option seemed optimal for our intentions.

Redundant traversals of the join tree. Another problem that presented itself during implementation was the redundant traversal of the join tree. Essentially, the whole join tree would have had to be traversed two times: The first traversal is needed in order to select a path or a subtree containing the relations that would eventually be joined to form a root guard. Then, upon having realized these joins, the whole tree would have been traversed again by AggJoin.

Due to the above mentioned practical problems that we encountered during the first effort of implementation, we decided to abandon the implementation of the *Upfront Joins* method. Instead, we opted to focus on implementing *PartAggJoin*, that integrates the necessary minimal materialization directly into the AggJoin procedure.

6.3 Implementation

In this chapter, we will present an algorithm that enforces guardedness for acyclic queries by allowing minimal necessary materialization. For the reasons discussed in Section 6.2, we eventually decided to fully implement an extension for AggJoin that closely follows the idea of *PartAggJoin*, which was presented in Section 6.2.2. The extension was implemented for SparkSQL Version 3.5.0.

6.3.1 The PartAggJoin Method

PartAggJoin makes it possible to integrate the necessary minimal materialization directly into the AggJoin procedure. As described above in Section 6.2.2, PartAggJoin makes a case-decision at every node in the join tree. If neither of the relations that need to be joined in the current node contains grouping attributes, AggJoin is selected. Otherwise,

PartAggJoin selects a "normal" join operator plus projection and aggregation, while additionally adding some attributes for frequency and aggregates. These additional attributes are needed, because just joining the relations is not enough. In order to make these joins compatible with AggJoin, a frequency attribute and (depending on the query) one or more aggregation attributes need to be initialized and propagated in order to perform the final aggregation at the root node later on.

The initialization of aggregates and frequency attributes is exactly the same as in AggJoin. Assuming that for every tuple t the frequency $t.c$ has been initialized as 1 and given an aggregate function of the form $A_j(f_j(\overline{B_j}))$ the additional attribute Agg_j is initialized as described below.

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

For the propagation of values, we have to consider two cases, depending on which form of join is performed in the current node: a join with full materialization, or an AggJoin. If a join with full materialization is performed, that is grouping attributes are propagated from the child node to the parent node, the case is simple: Since every tuple is materialized, it suffices to propagate the current values of c and Agg_j. Neither the frequency, nor the aggregation attribute changes.

The other case, an AggJoin, occurs when the child node does not contain grouping attributes. Then, we have to perform an AggJoin an update the values of c and Agg_j in the parent accordingly, that is $t.Agg_j := (\sum_{m=1}^{n} t_m.Agg_j[u_1])$. Note that in case of multiple children, $t.Agg_j := (\sum_{m=1}^{n} t_m.Agg_j[u_1]) * \prod_{i=2}^{k} s_i$. has to be computed.

Eventually, this results in a join tree whose subtrees will be evaluated by different methods: Subtrees without grouping attributes will be processed by AggJoin without any materialization, whereas relations of subtrees that do contain grouping attributes will be joined by ordinary join methods (plus the additional propagation of pass-through values for frequency and aggregation-values).

This approach solved both of the problems mentioned above: Since the relations are joined during the AggJoin procedure, there is no need to feed an intermediate result back to an early step in the processing pipeline - there simply is no intermediate result anymore. The two steps, namely creating a root guard and then processing the new join tree, are now conflated into one procedure. Naturally, the problem of redundant traversals was also resolved for the same reason.

To summarize, **"Partial AggJoin"** discriminates between the different subtrees of a join tree. All subtrees that do not contain grouping attributes that need to be propagated can

be processed by AggJoin and benefit from the resulting optimization. The other subtrees have to be processed without a Yannakakis-style optimization, but still add columns for frequency and aggregates, which allows for smooth integration with AggJoin.

In the next section, we will present a detailed example of the process described above.

6.3.2 Pseudo Code and Examples

The pseudo code of the approach described above can be seen in Algorithm 6.1. The inputs R, S, I_S and I_R are the same as for AggJoin: Assuming we want to join the relations R (parent) and S (child), List R and List S contain tuples that have join partners; List I_S contains aggregates that are present in both R and S; List I_R contains aggregates that are only present in R. An additional parameter is added in PartAggJoin: G_S that contains all grouping attributes that are present in S (the so-called applicable grouping attributes). Frequency attributes are assumed to be already initialized.

The procedure goes as follows: If G_S is empty (line 2), then we do not have to alter the procedure and can use AggJoin for optimization. Otherwise, we have to do a join with full materialization in order to propagate grouping attributes. To that end, all Agg_j values are initialized if the algorithm is in a leaf node (line 4 to line 9). Afterwards, the tuples in R and S are joined (line 11). In the next step, the aggregates are propagated. Note that within the SparkSQL query engine, projecting and propagating are done by one function. For better understanding, the pseudo code depicts the projection of attributes (line 12) and propagating attribute values (line 13 to line 25) as two separate steps. Finally, the frequency values are updated and R is returned. Note that R now holds the result of the join plus additional frequency and aggregate attributes.

6.3.3 Example (Theory)

As an example, we will evaluate the query in Figure 6.11. Note that we denote nodes of the join tree with parenthesis, e.g. (A) and relations and their attributes in italics, e.g. A. Since the query contains two distinct grouping attributes $A.a$ and $C.x$, it is neither guarded nor piecewise-guarded. Since one of the group guards A or C have to be the root node of the join tree, we chose to represent the query with the join tree depicted in Figure 6.12. We can differentiate between two subtrees: (A)-(B)-(C) has to be evaluated by performing joins with full materialization, since the grouping attribute $C.x$ has to be passed from (A) to (C). The subtree (B)-(D)-... on the other hand can be evaluated with AggJoin. Note that even if not depicted specifically, we assume that (D) has an arbitrary number of child nodes whose frequencies and aggregates have been propagated to node (D). Before starting to evaluate the join tree, the frequency and aggregate values are initialized. As described above in Section 4.5 and Section 4.3, all frequencies are initialized as 1. The aggregate, here SUM(z), is initialized as the value of z if z is present in a tuple and as 0 otherwise. For propagation, the values of SUM(z) is aliased as Agg_1.

Figure 6.13 shows the first join that is performed. In this case, a join with full materialization is needed, because relation C contains a grouping attribute that is not present in

Algorithm 6.1: Partial Aggregation Join

Input: Two lists R, S of tuples with the same values of the join attributes;
Input: List $I_S = \{s_1, \ldots, s_m\}$ of indices of aggregate attributes Agg_{s_i}, present in both R and S (to be updated in R);
Input: List $I_R = \{r_1, \ldots, r_n\}$ of indices of aggregate attributes Agg_{r_i}, present only in R (only frequency of joined tuples is considered);
Input: List G_S containing all applicable grouping attributes in S;

```
1  Function PartAggJoin(R, S, I_S, I_R, G_S):
2      if G_S.isEmpty then return AggJoin(R, S, I_S, I_R) ;
3      else
4          if S.isLeaf then
               // Initialize values
5              foreach s ∈ I_S do
6                  if A_s ∈ {MIN, MAX} then val_s ← init[s];
7                  else if A_s ∈ {SUM, COUNT} then val_s ← 0;
8              end
               // Add attributes to S
9              S.initialize(I_S, val_s);
10         end
           // Join R and S
11         R = Join(R, S);
           // Project c and Agg_i so that R has the same values as
               S for c and Agg_i
12         projected_tuples ← project(R, c, Agg_i);
13         foreach t ∈ projected_tuples do
14             if t exists in R then
                   // Propagate value from S to R for Agg_i
15                 foreach i ∈ I_S do
16                     if A_i = MIN then R[t][i] ← min(R[t][i], S[t][i]);
17                     else if A_i = MAX then R[t][i] ← max(R[t][i], S[t][i]);
18                     else if A_i = SUM then R[t][i] ← R[t][i] + S[t][i];
19                     else if A_i = COUNT then R[t][i] ← S[t][i] × R[t][i];
20                 end
21             end
               // Propagate c from S to R
22             foreach i ∈ I_R do
23                 R[t][i] ← R[t][i] × S[t].c;
24             end
25         end
26         return R;
27     end
```

```
SELECT A.a, C.x, SUM(z)
FROM A
NATURAL JOIN B
NATURAL JOIN C
NATURAL JOIN D
...
GROUP BY A.a, C.x
```

Figure 6.11: A query that is neither guarded nor piecewise-guarded. Note that "..." is a placeholder for one or more further joins that involve relation that hold no grouping attributes.

the root node (relation A). After joining, the intermediate result is $B1$, which contains the original tuples of B (cells with black border) plus new tuples and attributes (cells with green border). Naturally, tuples in B that did not have a join partner in C have been eliminated.

The next step is shown in Figure 6.14. In this case the child node D does not contain any grouping attributes, so an AggJoin is performed. Note that D contains the propagated frequencies and aggregates of its subtree that have already been evaluated with AggJoin. As a result of $D \bowtie_{AggJoin} B$ we obtain $B2$. No new tuples were added and the frequency values as well as the values of Agg_1 have been updated accordingly. Again, tuples in B that have no join partner in D are eliminated.

The results of the final join of the algorithm can be seen in Figure 6.15. Again, we have to perform a join with full materialization in order to propagate the missing grouping attribute $C.x$ to relation A. The already computed aggregates and frequencies are simply passed on without further calculations. Now that all aggregates and grouping attributes are present in root, the result can be calculated and unnecessary columns can be projected away. Figure 6.16 shows the result of our example.

6.3.4 Example (Practice)

What PartAggJoin looks like in practice can be best shown by comparing the logical query plans that are produced by AggJoin and PartAggJoin. For spatial reasons, the right part of the plans is cropped in this section, however, the important information, namely the join operator, is highlighted in colors.

At first, we will look at a regular AggJoin evaluation. For this example, we chose the query *q3* of the Syn-TPC-H dataset (also seen in Figure 6.17). Figure 6.18 shows the corresponding logical query plan in which one can see that the AggJoin operator (named *CountJoin* in the query plan for legacy reasons) was applied instead of a regular join operator.

For comparison, let us now look at an example of PartAggJoin. We evaluate q34 of the Syn-TPC-H dataset (Figure 6.19) and obtain the logical query plan that can be seen

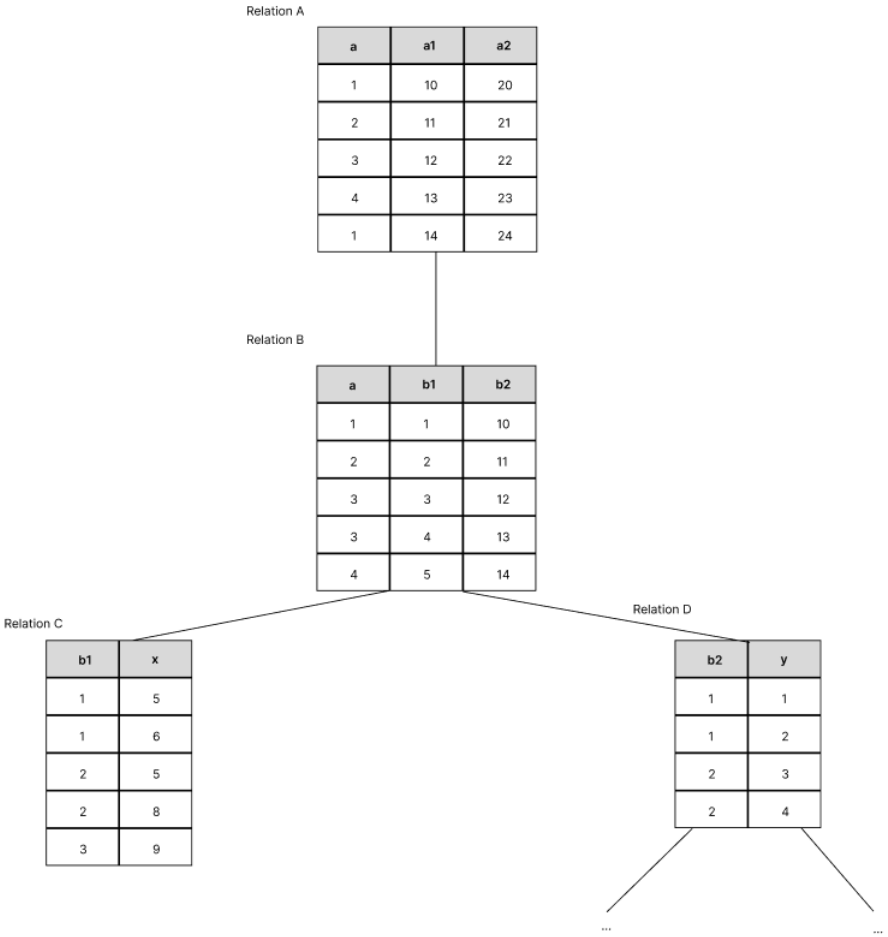


Figure 6.12: The initial join tree for the query in Figure 6.11. Note that "..." is a placeholder for an arbitrary number of child nodes that hold no grouping attributes.

in Figure 6.20. Note that q34 is a query that involves five joins. Since there are two grouping attributes that stem from different relations, exactly one AggJoin operator (CountJoin) is substituted with a regular join with full materialization.

We note that there is a slight difference between the theoretical example presented above and the practical example shown in this section. In the theoretical example, *all* joins from the root node to the relation(s) that contain grouping attributes have been substituted with regular joins. But if one looks closely at Figure 6.20, one can see that only one join was substituted and the root of the logical query plan was not part of the join. This is because SparkSQL applies its own optimizations to the logical query plan on top of AggJoin or PartAggJoin which might involve rotating the original query tree and changing the root node. However, the grouping attributes are still preserved by applying projection (and aggregation) to the relevant attributes.

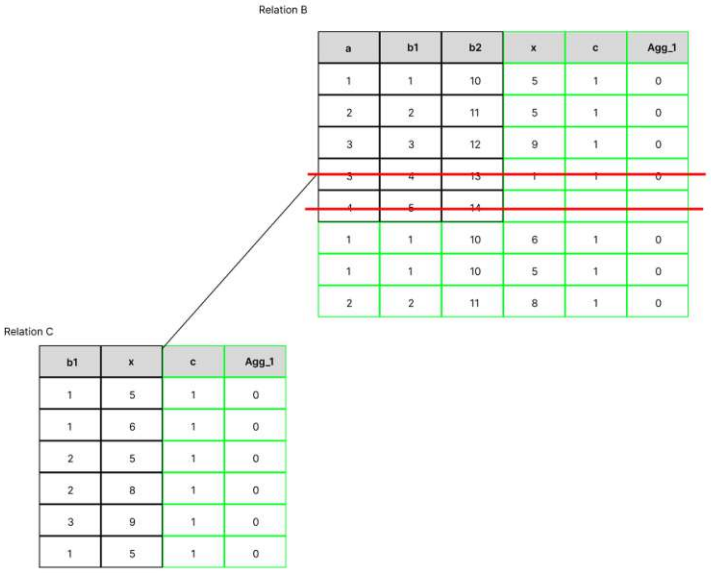


Figure 6.13: The first intermediate result $B1 = B \bowtie C$ is obtained by full materialization of a join. Cells that have been added as a result of the join have a green border.

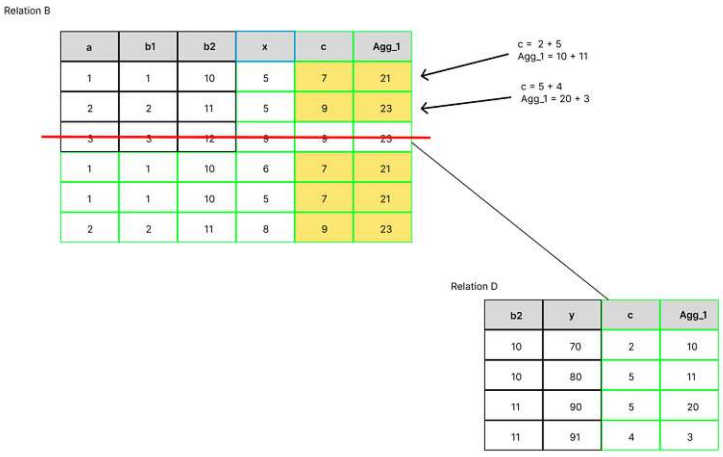


Figure 6.14: The second intermediate result is $B2 = D \bowtie_{AggJoin} B1$. Since D does not hold any grouping attributes, AggJoin can be applied. Note that D holds the propagated frequencies and aggregates of the children of node (D).

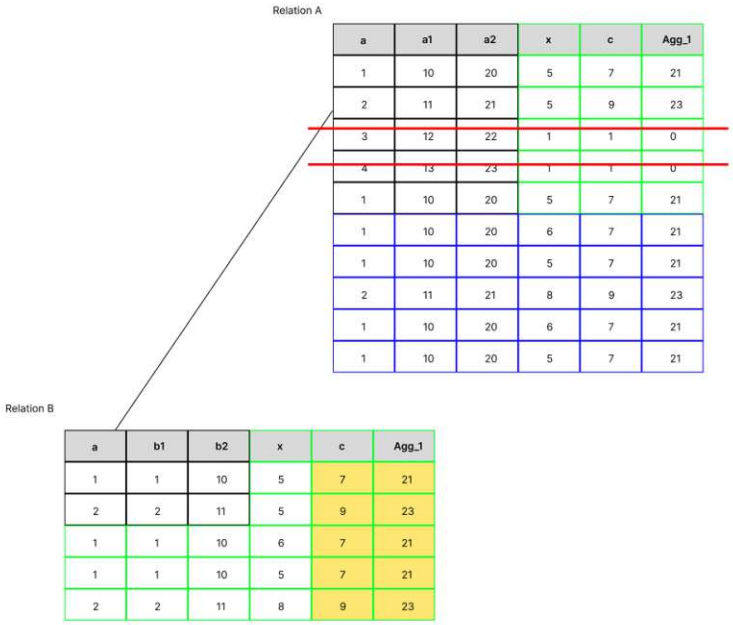


Figure 6.15: After the final join, relation A contains all aggregates and grouping attributes needed to evaluate the query.

a	x	Agg_1
1	5	84
2	5	23
1	6	42
2	8	23

Figure 6.16: The final result of the query in Figure 6.11, evaluated on the join tree depicted in Figure 6.12.

```
SELECT
    n_name AS Nation,
    SUM(l_extendedprice * (1 - l_discount)) AS Total_Spending
FROM
    customer
JOIN
    orders
    ON customer.c_custkey = orders.o_custkey
JOIN
    lineitem
    ON orders.o_orderkey = lineitem.l_orderkey
JOIN
    nation
    ON customer.c_nationkey = nation.n_nationkey
GROUP BY
    n_name
ORDER BY
    Total_Spending DESC;
```

Figure 6.17: Example of a piecewise-guarded query (q3 of the Syn-TPC-H dataset)

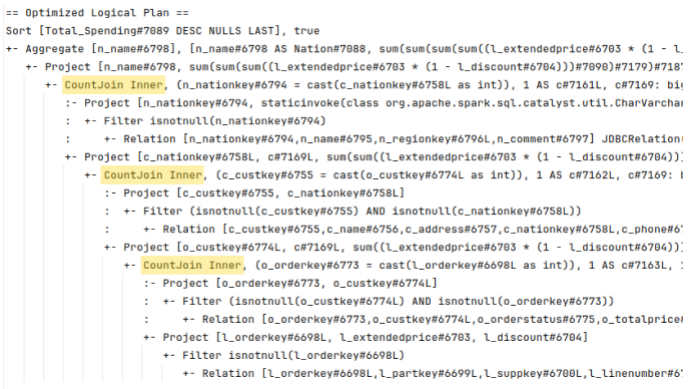


Figure 6.18: The logical query plan of the query in Figure 6.17 containing the AggJoin operator (depicted in the image as *CountJoin* for legacy reasons), highlighted in yellow

```
SELECT
    s_nationkey AS Nation_Key,
    p_type AS Part_Type,
    MIN(ps_supplycost) AS Min_Supply_Cost
FROM
    supplier
JOIN
    nation ON s_nationkey = n_nationkey
JOIN
    region ON n_regionkey = r_regionkey
JOIN
    partsupp ON s_suppkey = ps_suppkey
JOIN
    part ON ps_partkey = p_partkey
JOIN
    lineitem ON p_partkey = l_partkey
GROUP BY
    s_nationkey, p_type;
```

Figure 6.19: Example of a non-guarded query (q34 of the Syn-TPC-H dataset)

6.3.5 Best and Worst Cases

As one can see in the example above, *PartAggJoin* widens the field of application for AggJoin by evaluating subtrees without grouping attributes with AggJoin and performing joins with full materialization for subtrees that contain grouping attributes. Before this optimization, there was no possibility to optimize non-guarded queries with AggJoin. Applying *PartAggJoin* method now allows a more modular approach: The processing of subtrees that are guarded can be optimized by AggJoin, and only subtrees that are non-guarded have to utilized joins with full materialization. Thus, one goal of the thesis is achieved: non-guarded queries are now eligible for processing with AggJoin.

However, whether the speed-gain achieved by AggJoin can outweigh the overhead induced by full materialization (even though it is still the minimal necessary materialization) depends on the join tree at hand. Assuming that relation A and relation B are relatively small relations and furthermore the only ones that contain grouping attributes, Figure 6.21 shows an example of an optimal join tree. Even if (D) had more child nodes, one could always enforce guardedness by performing only one cheap join.

A worst case example can be seen in Figure 6.22. Assuming that the grouping attributes are contained in relation A, D and E, every single join has to be materialized in order to propagate every grouping attribute to the root node that holds relation A. Because of the suboptimal distribution of grouping attributes among the relations, the optimization that AggJoin provides cannot be utilized.

```
== Optimized Logical Plan ==
Aggregate [s_nationkey#22581L, p_type#22644], [s_nationkey#22581L AS Supplier_Account_Balance#22811L, p_type#22644 AS Part_Type#22812, min(min(ps_s
+- Project [s_nationkey#22581L, p_type#22644, agg#22814, s_nationkey#22581L, p_type#22644]
   +- CountJoin Inner, (s_nationkey#22581L = cast(n_nationkey#22700 as bigint)), c#22911: bigint, c#22920: bigint
      :- Aggregate [s_nationkey#22581L, p_type#22644], [s_nationkey#22581L, p_type#22644, min(ps_supplycost#22597) AS agg#22814, 1 AS c#22911L, s_n
      :  +- Project [s_nationkey#22581L, ps_supplycost#22597, p_type#22644]
      :     +- Join Inner, (s_suppkey#22578 = cast(ps_suppkey#22595L as int))
      :        :- Project [s_suppkey#22578, s_nationkey#22581L]
      :        :  +- Filter (isnotnull(s_nationkey#22581L) AND isnotnull(s_suppkey#22578))
      :        :     +- Relation [s_suppkey#22578,s_name#22579,s_address#22580,s_nationkey#22581L,s_phone#22582,s_acctbal#22583,s_comment#22584] JD
      :        +- Project [ps_suppkey#22595L, ps_supplycost#22597, p_type#22644]
      :           +- Join Inner, (ps_partkey#22594L = cast(p_partkey#22640 as bigint))
      :              :- Project [ps_partkey#22594L, ps_suppkey#22595L, ps_supplycost#22597]
      :              :  +- CountJoin Inner, (ps_partkey#22594L = cast(l_partkey#22605L as bigint)), 1 AS c#22888L, 1 AS c#22894L
      :              :     :- Project [ps_partkey#22594L, ps_suppkey#22595L, ps_supplycost#22597]
      :              :     :  +- Filter (isnotnull(ps_suppkey#22595L) AND isnotnull(ps_partkey#22594L))
      :              :     :     +- Relation [ps_partkey#22594L,ps_suppkey#22595L,ps_availqty#22596,ps_supplycost#22597,ps_comment#22598] JDBCRelat
      :              :     +- Project [l_partkey#22605L]
      :              :        +- Filter isnotnull(l_partkey#22605L)
      :              :           +- Relation [l_orderkey#22604L,l_partkey#22605L,l_suppkey#22606L,l_linenumber#22607,l_quantity#22608,l_extendedpri
      :              +- Project [p_partkey#22640, p_type#22644]
      :                 +- Filter isnotnull(p_partkey#22640)
      :                    +- Relation [p_partkey#22640,p_name#22641,p_mfgr#22642,p_brand#22643,p_type#22644,p_size#22645,p_container#22646,p_retai
      +- Project [n_nationkey#22700, c#22920L]
         +- CountJoin Inner, (n_regionkey#22702L = cast(r_regionkey#22709 as bigint)), 1 AS c#22914L, 1 AS c#22920L
            :- Project [n_nationkey#22700, n_regionkey#22702L]
            :  +- Filter (isnotnull(n_nationkey#22700) AND isnotnull(n_regionkey#22702L))
            :     +- Relation [n_nationkey#22700,n_name#22701,n_regionkey#22702L,n_comment#22703] JDBCRelation(nation) [numPartitions=1]
            +- Project [r_regionkey#22709]
               +- Filter isnotnull(r_regionkey#22709)
                  +- Relation [r_regionkey#22709,r_name#22710,r_comment#22711] JDBCRelation(region) [numPartitions=1]
```

Figure 6.20: The logical query plan of the query in Figure 6.19 containing both the AggJoin operator (*CountJoin Inner*, highlighted in yellow), and the normal join operator (*Join Inner*, highlighted in orange) as part of the PartAggJoin procedure.

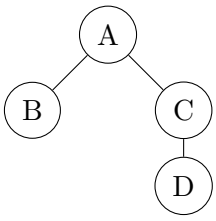


Figure 6.21: An example of an optimal case for partial AggJoin: Assume that A and B are small relations and the only ones that contain grouping attributes.

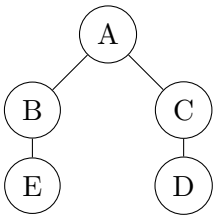


Figure 6.22: Assume that the grouping attributes of a query are contained in the relations A, D and E. In this worst case scenario, the whole join tree has to be evaluated by materializing every single join.

These two examples highlight the need for a way to categorize a query and its necessary joins before evaluating it in some way. Even in the optimal case, the algorithm would greatly benefit from cardinality estimation, but this is a matter of Spark's internal development. Another option would be to leverage machine learning (ML) in order to extract key features of a query that would enable the algorithm to quickly decide whether a query benefits from partial AggJoins or not. More details will be discussed in the outlook given in Chapter 8.

6.4 Evaluation

6.4.1 Experimental Setup

We perform the experiments on a machine running on Linux Mint 21 Cinnamon with 13,5GB RAM and and the AMD Ryzen 7 7730U with Radeon Graphics × 8 processor. Furthermore, our extension was implemented for SparkSQL version 3.5.0. Our implementation of PartAggJoin has been published on GitHub (https://github.com/A-Ortner/master-thesis-project). The setup for the executed benchmarks can be found here: https://github.com/A-Ortner/master-thesis-benchmark. The goal of our experiments are the following:

1. Goal 1: Verify whether PartAggJoin can enforce guardedness for queries that could not be processed by AggJoin by comparing AggJoin and PartAggJoin.

2. Goal 2: Verify whether PartAggJoin accurately discriminates between non-guarded and (piecewise-)guarded queries and only uses joins with full materialization when it is necessary (thus guaranteeing *minimal* materialization).

3. Goal 3: Verify whether PartAggJoin can enforce guardedness efficiently by comparing PartAggJoin to the original SparkSQL distribution.

To obtain meaningful results from our evaluation, we chose the benchmark datasets TPC-H and Syn-TPC-H. We refrained from using JOB, STATS, SNAP and LSQB because they contain no non-guarded queries. TPC-DS seemed more suitable, but since it contains very complex queries that could also not be fully processed by AggJoin, we decided that it held little comparative value for PartAggJoin, since PartAggJoin would fail for reasons other than non-guardedness. This left us with TPC-H, which contained some non-guarded queries that PartAggJoin should be able to evaluate, whereas AggJoin could not. In order to specifically test our implementation on a bigger set of non-guarded queries, we created Syn-TPC-H (see Section 6.1.3) and compared how AggJoin, PartAggJoin and the original SparkSQL query engine performed during evaluation. Note that for legacy reasons, the presented logical query plan uses the name *CountJoin* when the AggJoin operator is used.

benchmark	#queries	⋈-agg	acyc.	AggJoin	PartAggJoin
TPC-H	22	15	14	7	12
Syn-TPC-H	45	45	45	10	45

Table 6.1: Overview of applicability of AggJoin and PartAggJoin to the benchmarks Syn-TPC-H and TPC-H. The columns mark the number of queries (#queries), the number of aggregate queries (⋈-agg), the number of acyclic queries (acyc.) and the number of queries that could be evaluated by the two methods AggJoin and PartAggJoin.

6.4.2 Enforcing Guardedness

For the first part of the evaluation, we wanted to find out whether PartAggJoin was able to enforce guardedness for queries that could not be evaluated by AggJoin. To this end, we executed the non-guarded queries of the Syn-TPC-H dataset and the non-guarded queries of the original TPC-H dataset with AggJoin and PartAggJoin. Our evaluation has shown that PartAggJoin was able to evaluate every non-guarded query in the Syn-TPC-H dataset and more than half of the non-guarded queries in the original TPC-H dataset. More details can be found in Table 6.1. Note that the Syn-TPC-H was specifically designed with a focus on acyclic aggregate queries, which is why the overall coverage for any Yannakakis-style optimization is much better.

As we can see in Table 6.1, PartAggJoin was able to raise the coverage of the Syn-TPC-H dataset from 0% (if only AggJoin is enabled) to 100%, which greatly increased the applicability of Yannakakis-style optimization for this dataset. When it comes to TPC-H the overall coverage is lower because only 14 queries are acyclic and thus eligible for Yannakakis-style optimization. Of these 14 queries, AggJoin is able to evaluate 7 queries, whereas PartAggJoin increased the number of eligible queries to 12, which almost doubles the amount of ACQs that can be evaluated. We will discuss the possible reasons as to why some queries could neither be evaluated by AggJoin nor PartAggJoin in Chapter 7.

6.4.3 Minimal Materialization

After verifying in the section above that PartAggJoin is indeed able to enforce guardedness for non-guarded queries, the next step is to check whether the materialization that is necessary for this procedure is indeed minimal. This is the case due to the way PartAggJoin was implemented. The AggJoin operator is only substituted if applicable grouping attributes are present in the relations that are to be joined at that step of the query evaluation, otherwise AggJoin will be applied. As an example, let us look at the query in Figure 6.23, which corresponds to q22.sql in the Syn-TPC-H dataset. This query has two grouping attributes that stem from different relations, so at least one AggJoin has to be substituted by a join with materialization. In the corresponding logical plan, depicted in Figure 6.24, we can see that this is indeed the case. During our evaluation, we could verify for all queries in the Syn-TPC-H benchmark, that the materialization that occurs is indeed minimal.

```
SELECT
    r_name AS Region,
    n_name AS Nation,
    COUNT(DISTINCT s_suppkey) AS Supplier_Count,
    MIN(ps_supplycost) AS Min_Supply_Cost,
    MAX(ps_availqty) AS Max_Available_Quantity
FROM
    region
JOIN
    nation ON r_regionkey = n_regionkey
JOIN
    supplier ON n_nationkey = s_nationkey
JOIN
    partsupp ON s_suppkey = ps_suppkey
GROUP BY
    r_name, n_name
ORDER BY
    Supplier_Count DESC;
```

Figure 6.23: Non-guarded query that should be evaluated by allowing one join with full materialization of the relations *region* and *nation*. The query corresponds to q22.sql of the Syn-TPC-H dataset.

```
== Optimized Logical Plan ==
Sort [Supplier_Count#10430L DESC NULLS LAST], true
+- Aggregate [r_name#10329, n_name#10321], [r_name#10329 AS Region#10428, n_name#10321 AS Nation#10429, sum(sum#10546L)
   +- Project [n_name#10321, r_name#10329, min(min(ps_supplycost#10214)#10434)#10434, max(max(ps_availqty#10213)#10435)#
      +- CountJoin Inner, (n_nationkey#10317 = cast(s_nationkey#10198L as int)), c#10527: bigint, c#10536: bigint, [coun
         :- Project [n_nationkey#10317, n_name#10321, r_name#10329, 1 AS c#10527L]
         :  +- Join Inner, (n_regionkey#10319L = cast(r_regionkey#10326 as bigint))
         :     :- Project [n_nationkey#10317, staticinvoke(class org.apache.spark.sql.catalyst.util.CharVarcharCodegenU
         :     :  +- Filter (isnotnull(n_regionkey#10319L) AND isnotnull(n_nationkey#10317))
         :     :     +- Relation [n_nationkey#10317,n_name#10318,n_regionkey#10319L,n_comment#10320] JDBCRelation(natio
         :     +- Project [r_regionkey#10326, staticinvoke(class org.apache.spark.sql.catalyst.util.CharVarcharCodegenU
         :        +- Filter isnotnull(r_regionkey#10326)
         :           +- Relation [r_regionkey#10326,r_name#10327,r_comment#10328] JDBCRelation(region) [numPartitions=1
         +- CountJoin Inner, (s_suppkey#10195 = cast(ps_suppkey#10212L as int)), 1 AS c#10530L, 1 AS c#10536L, [min(ps_s
            :- Project [s_suppkey#10195, s_nationkey#10198L]
            :  +- Filter (isnotnull(s_nationkey#10198L) AND isnotnull(s_suppkey#10195))
            :     +- Relation [s_suppkey#10195,s_name#10196,s_address#10197,s_nationkey#10198L,s_phone#10199,s_acctbal#1
            +- Project [ps_suppkey#10212L, ps_availqty#10213, ps_supplycost#10214]
               +- Filter isnotnull(ps_suppkey#10212L)
                  +- Relation [ps_partkey#10211L,ps_suppkey#10212L,ps_availqty#10213,ps_supplycost#10214,ps_comment#1021
```

Figure 6.24: (Optimized) Logical plan of the query shown in Figure 6.23. As one can see, PartAggJoin allows for exactly one join (written as *Join, Inner*, marked in orange), whereas all other joins are evaluated by the AggJoin operator (here named *CountJoin*). For spatial reasons, the image was cropped.

6.4.4 Performance

After having seen that PartAggJoin is able to enforce guardedness and does so by only allowing minimal materalization, the last question of our evaluation is whether the process is an optimization in comparison to the original SparkSQL processing. To answer this question, we used the non-guarded queries of the Syn-TPC-H dataset and evaluated them with PartAggJoin and the original SparkSQL query engine. Due to statistical accuracy we executed the benchmark five times and averaged the runtime for each query. The results can be seen in Figure 6.25.

Overall, PartAggJoin was faster than the original algorithm in 21 out of 35 cases (60%). On average, it was 1.99 seconds faster than SparkSQL. In 14 out 35 cases, PartAggJoin could not speed up the evaluation of the given query. However, even in these cases it was on average only 0.80 seconds slower than the original implementation, which can be attributed to the fact that the more costly a query is, the more optimization can be gained by using PartAggJoin.

Another metric that is worth mentioning is the end-to-end runtime (see Figure 6.26). If we compare the time it took both systems (PartAggJoin and the original SparkSQL distribution) to evaluate the non-guarded queries of the Syn-TPC-H dataset, we can see that PartAggJoin achieves an overall speed-gain of 10%. In concrete numbers, SparkSQL took 5 minutes and 23,78 seconds to evaluate Syn-TPC-H, whereas PartAggJoin took only 4 minutes and 53,44 seconds which amounts to an absolute speedgain of 30,334 seconds. On average it took PartAggJoin 0.89 seconds less time to evaluate the non-guarded queries.

We conclude this section by acknowledging that PartAggJoin is able to enforce guardedness with minimal materialization and additionally achieves speed-gains in comparison to standard SparkSQL. In the next section, we will discuss why, in some cases, no optimization occurred and other limitations of PartAggJoin.

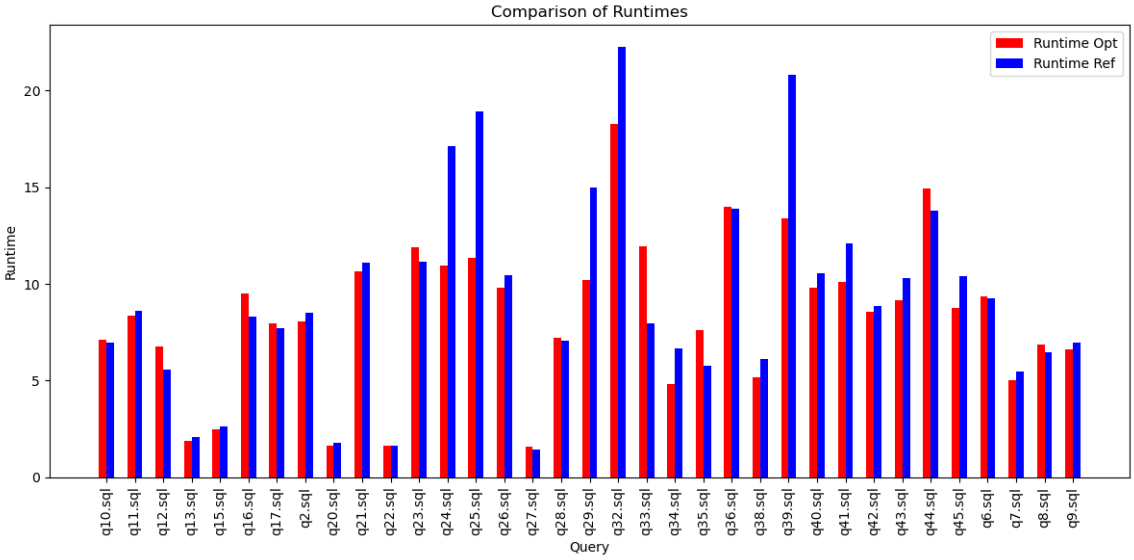


Figure 6.25: Comparison of runtimes for the non-guarded queries in the Syn-TPC-H dataset. In red, the runtime of our optimization (PartAggJoin) is depicted, the blue bars indicate the runtime of the original SparkSQL module.

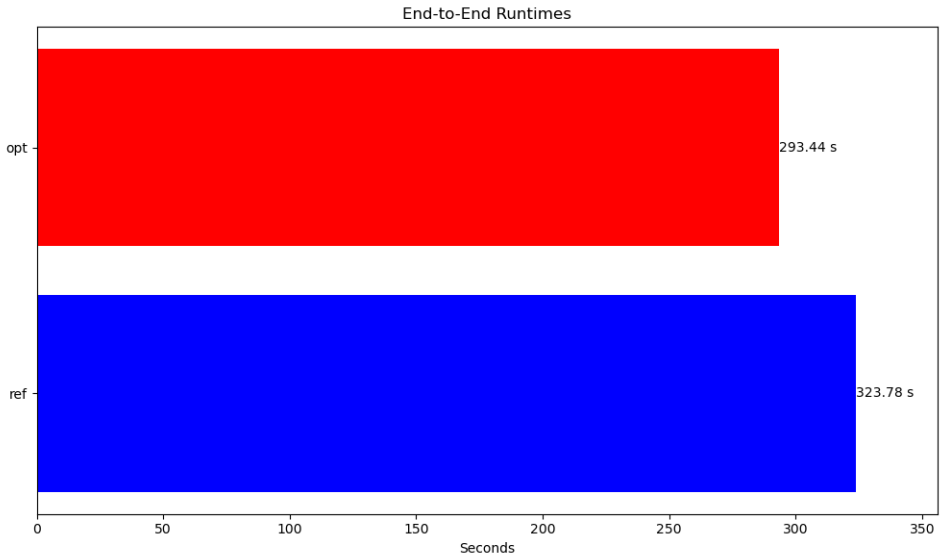


Figure 6.26: Comparison of the end-to-end for the non-guarded queries in the Syn-TPC-H dataset. In red ("opt"), the runtime of our optimization (PartAggJoin) is depicted, the blue bar ("ref") indicates the end-to-end runtime of the original SparkSQL module.

CHAPTER 7

Discussion and Results

In this section, we will revisit the research questions that motivated this work and discuss the results obtained in Section 6.4.

The first research question is **"How can we algorithmically restructure non-guarded ACQs into guarded ACQs in a both time and space efficient way in order to make them eligible for the query optimization proposed in [LPS24]?"** To answer this question, we compared the two methods *Upfront Joins* and *PartAggJoin*. As has been discussed in Section 6.2, we found that *Upfront Joins* is not an ideal method for a Spark environment, since the feedback mechanisms available between the different stages of query processing are only rudimentary present and do not suffice for a method like *Upfront Joins*. Furthermore, *PartAggJoin* is a direct extension of AggJoin and thus avoids the additional traversals of the join tree that are necessary for the *Upfront Joins* method. We conclude, that *PartAggJoin* is a suitable method to algorithmically transform non-guarded queries so that they can be processed with Yannakakis-style optimization techniques (here: AggJoin) in place.

The second research questions is **"How can we extend the state-of-the-art implementation of AggJoin presented in [LPS24] for query optimization to make it more applicable to real-world scenarios?"**

To answer this question, we will take a closer look at the evaluation result obtained in Section 6.4.

Overall, our implementation did perform well: It was able to successfully transform all non-guarded queries in the Syn-TPC-H dataset and more than half of the non-guarded queries in TPC-H into guarded queries. Furthermore, PartAggJoin was able to correctly discern guarded from non-guarded queries and only allowed materialization where it was necessary. When it comes to performance, it was able to speed-up query processing in 60% of our test cases in comparison to the original implementation of the SparkSQL module.

For the rest of this section, we will discuss the limitations of our implementation that have been revealed by our benchmarks. Firstly, we will speculate on why some non-guarded queries of the TPC-H dataset could not be processed. Secondly, we will take a closer look at the runtime comparison between the original SparkSQL module an PartAggJoin.

Enforcing Guardedness

In our benchmarks, PartAggJoin was able to process 57 out of 59 non-guarded queries, which is 96.6%. Even though these are exciting results in terms of applicability of our implementation, we want to discuss the fact that 2 queries of the TPC-H dataset (q3 and q9) could not be evaluated. We note that the given explanations remain partly speculative because the inner heuristics that SparkSQL applies are very complex and not easily traceable.

One possible explanation is that the query structure itself might be too complex. One of the queries (q9) involves 6 relations and contains a subquery. It could be that this structure might not be recognized either by PartAggJoin or even AggJoin (since there are also some piecewise-guarded ACQs that cannot be processed by AggJoin in the first place).

Another explanation regards a certain flakey behavior of the Catalyst optimizer that might occur due to internal heuristics. In some cases, a query was only evaluated in some runs, but could not be processed in other iterations. This might be because the optimizer applies certain optimizations on top of PartAggJoin (such as projecting away seemingly unnecessary columns) that then alters the logical and/or physical join plan, which in turn interferes with the intended purpose of PartAggJoin.

Performance

As described in Section 6.4, we compared the runtime of PartAggJoin with the original SparkSQL implementation by benchmarking the systems on the Syn-TPC-H dataset.

PartAggJoin was able to out-class the original implementation in 60% of all cases, with an average speed-gain of 1.99 seconds. We also note that the longer query evaluation takes for a certain query, the bigger the speed up. The biggest speed-up was measured regarding q39, where PartAggJoin beat SparkSQL by 7.42 seconds. The biggest loss occurred for q33, where the original implementation was by 3.99 seconds faster than PartAggJoin. One example, where both implementations took almost the same time to process the query, was q22 with a runtime difference of only 0.005 seconds. To understand the performance results of PartAggJoin, we will now take a closer look at these three queries and discuss their logical query plans. Note that the explanations given in the following paragraphs are partly based on educated guesses since Spark does not offer advanced cardinality estimation.

Firstly, let us compare the static features of the three queries q22, q33 and q33 in Table 7.1. Since both q39, the query where PartAggJoin offered the biggest speed-gain,

-	#relations	#aggregate functions	#grouping attributes
q22	4	3	2
q33	6	1	2
q39	6	1	2

Table 7.1: Comparing the features of the queries q22, q33 and q39 of the benchmark Syn-TPC-H.

and q33, where PartAggJoin performed poorest, we conclude that the critical difference must be found elsewhere. The original queries can be found in Appendix B.

To look deeper into this problem, let us compare the logical query plans of query q22, q33 and q39 of the Syn-TPC-H dataset. In this section, we will use simplified query plans; the original query plans can be found in Appendix B.

According to the benchmarks, q22 has roughly the same runtime for both PartAggJoin and SparkSQL. The simplified query plan shown in Figure 7.1 corresponds to the query plan in Figure B.4 in Appendix B. We can see that PartAggJoin only forces one join, whereas all other joins are taken care of by the AggJoin operator. One possible explanation for the almost equal runtimes could be that the relations N and R are the biggest relations of the query. If the other relations are fairly small, joining them with a regular join instead of AggJoin will not offer much benefit. And since the (presumably) most costly join, namely $N \bowtie R$ is executed with full materialization in both implementations, the runtime will be roughly equal. Furthermore, q22 involves few joins than the other two queries and the fewer joins are necessary, the more accurate are SparkSQL's internal heuristics. Consequently, the speed-up provided by AggJoin is matched by the speed-up gained by heuristics.

A more interesting insight offer query plans of q33 and q39. The simplified query plan shown in Figure 7.2 and Figure 7.3 correspond to the query plans depicted in Figure B.5 and Figure B.6 in Appendix B.

While the cardinalities of the involved relations also do play a role in these two instances, we can also identify a structural problem: q33 does take very long, even though PartAggJoin substitutes only one AggJoin with a full join. However, since it is the "root" join, it has to join all attributes that have been propagated thus far by the AggJoins in the subtrees. If we now further assume that some optimizations of the original SparkSQL module are not available because PartAggJoin forces the logical plan into this structure, it becomes apparent why PartAggJoin was not able to leverage its optimization potential in this case..

A completely different logical plan can be seen in case of q39. The root join is still an AggJoin, so the heaviest lifting is delegated to Yannakakis-style optimization. Furthermore, if we assume that the relations PS and P are relatively small, the subtree that is evaluated by join with materialization does not involve a lot of cost. In this case, PartAggJoin offers a big speed-gain in comparison to the original optimization, that

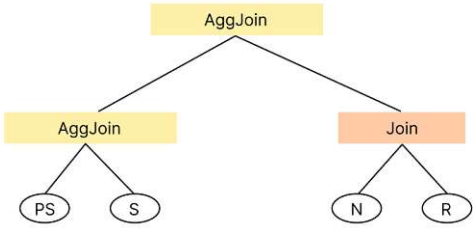


Figure 7.1: Simplified logical query plan of q22.

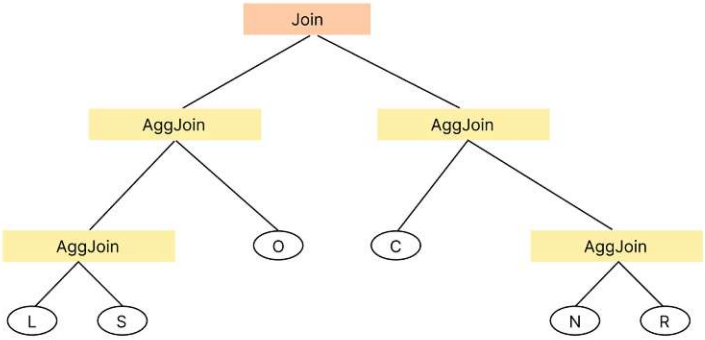


Figure 7.2: Simplified logical query plan of q33.

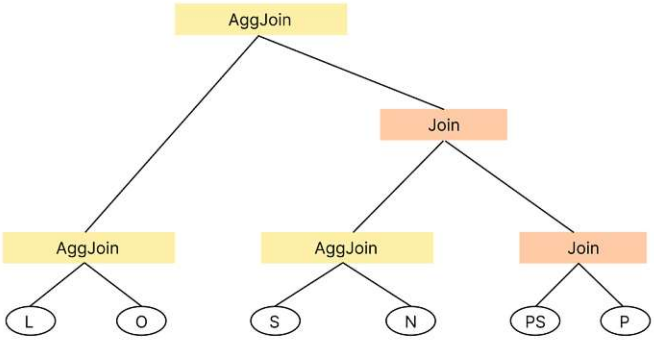


Figure 7.3: Simplified logical query plan of q39.

would have to materialize 3 more expensive joins.

This analysis shows once more, that PartAggJoin offers great optimization, but would immensely benefit from some sort of cardinality estimation.

CHAPTER 8

Conclusion

In this thesis, we sought to find a method to apply Yannakakis-style optimization to non-guarded queries. To this end, we presented three possible methods: *Upfront Joins*, *PartAggJoin* and *GroupAggJoin*. Since *GroupAggJoin* is currently being developed by the authors of [LPS24], the focus of this thesis was to find out which approach - *Upfront Joins* or *PartAggJoin* - is more suitable in an SparkSQL environment.

Upon comparing the two methods in theory and in practice, we came to the conclusion that *PartAggJoin* is a better suited approach for a Spark environment, since it better utilizes the given program structure of Spark's Catalyst query optimizer. On top of that, it avoids unnecessary traversals of a given logical plan in order to sort out relations that contain applicable grouping attributes. Our implementation was able to evaluate all non-guarded queries of our new benchmark dataset and more than half of the non-guarded queries in the TPC-H dataset. It successfully transforms non-guarded queries to queries that can be evaluated by Yannakakis-style optimization by splitting the join treeinto subtrees. If a subtree does not contain applicable grouping attributes, it can be evaluated by AggJoin [LPS24]. Should a subtree contain applicable grouping attributes, minimal necessary materialization is chosen for this subtree. This way, a query that was originally not able to be evaluated by AggJoin can now still be partially evaluated by this optimization and thus profit from the speed-gain obtained by this method. Since *PartAggJoin* is an extension for AggJoin, it is applicable for the same aggregate functions as AggJoin, namely MIN, MAX, SUM, COUNT and AVG.

The evaluation of PartAggJoin has shown that our implementation is able to transform non-guarded queries into guarded queries and thus make them eligible for Yannakakis-style optimization. Furthermore, we have also shown that it does this by only allowing *minimal* necessary optimization while still outperforming the original SparkSQL module in 60% of the cases evaluated. The comparison of the end-to-end runtimes of PartAggJoin and the original SparkSQL distribution showed an overall speed-gain of 10% over the non-

guarded queries of the Syn-TPC-H dataset, which means that on average PartAggJoin sped up the processing of each query by 0.89 seconds.

As has been noted in Section 6.3 and Chapter 7, there are some worst case instances of join trees that will not profit greatly from our extension. In particular, these worst cases are instances where a) AggJoin is rarely applied due to the distribution of grouping attributes in the join tree, b) the joins to be performed are very costly and c) the query involves complex query structures where SparkSQL's internal optimizations interfere with the measures of PartAggJoin.

In conclusion, our work was able to answer the two research questions that were proposed:

- How can we algorithmically restructure non-guarded ACQs into guarded ACQs in a both time and space efficient way in order to make them eligible for the query optimization proposed in [LPS24]? **PartAggJoin is a method that successfully allows non-guarded ACQs to be evaluated by Yannakakis-style optimization while still minimizing materialization.**
- How can we extend the state-of-the-art implementation of AggJoin presented in [LPS24]for query optimization to make it more applicable to real-world scenarios? **Our implementation of PartAggJoin as described in Section 6.3 has extended AggJoin by splitting a query tree into subtrees that are either processed by AggJoin or (if applicable grouping attributes are present) by joins with materialization. This has widened the field of application for Yannakakis-style optimization insofar that more than half of the non-guarded queries of the TPC-H dataset and all of the non-guarded queries in the Syn-TPC-H dataset could be evaluated that previously could not be processed by AggJoin.**

As future work, there are two possibilities to extend the research of this paper. One possibility would be to implement *PartAggJoin* in other DBMS like like PostgreSQL [1] or Oracle [2] which, unlike Spark, offer rich and advanced cardinality estimation. In such systems, one could apply a simple cost-based heuristic that decides whether *PartAggJoin* should be applied or not. The other option is to leverage machine learning to extract features from queries (like query structure, number of grouping attributes, etc.) in order to deduce the magnitude of optimization a query might gain by *PartAggJoin* in a SparkSQL environment. Both approaches will reduce the limitations identified in Section 6.4 and Chapter 7. It would also be interesting to compare GroupAggJoin and PartAggJoin in future evaluations, to see whether GroupAggJoin offers even more speed-gain despite involving a bigger computational overhead.

As of now, we have shown that Yannakakis-style optimization is not restricted to (piecewise-)guarded queries and have also shown a way to reduce the cost of query evaluation in future systems.

[1] https://www.postgresql.org/docs/current/row-estimation-examples.html
[2] https://blogs.oracle.com/optimizer/post/cardinality-and-dynamic-statistics

APPENDIX A

Syn-TPC-H

In this table, a detailed overview of all queries in the created Syn-TPC-H dataset is given.

ID	#relations	#aggregates	#grouped	method
1	2	2	1	AggJoin
2	2	1	2	PartAggJoin
3	4	1	1	AggJoin
4	4	2	1	AggJoin
5	5	2	1	AggJoin
6	2	2	2	PartAggJoin
7	2	1	2	PartAggJoin
8	2	1	2	PartAggJoin
9	2	1	2	PartAggJoin
10	2	1	2	PartAggJoin
11	3	1	2	PartAggJoin
12	3	1	2	PartAggJoin
13	3	1	2	PartAggJoin
14	4	1	1	AggJoin
15	2	1	2	PartAggJoin
16	2	1	2	PartAggJoin
17	3	1	2	PartAggJoin
18	5	1	1	AggJoin
19	3	1	1	AggJoin
20	3	1	2	PartAggJoin
21	3	1	2	PartAggJoin
22	4	3	2	PartAggJoin
23	2	2	2	PartAggJoin
24	2	3	2	PartAggJoin
25	2	4	2	PartAggJoin
26	3	3	2	PartAggJoin
27	4	2	2	PartAggJoin
28	2	1	2	PartAggJoin
29	2	3	2	PartAggJoin
30	6	2	2	AggJoin
31	6	2	2	AggJoin
32	6	1	2	PartAggJoin
33	6	1	2	PartAggJoin
34	6	1	2	PartAggJoin
35	6	1	2	PartAggJoin
36	3	4	2	PartAggJoin
37	6	1	1	AggJoin
38	6	1	2	PartAggJoin
39	6	1	2	PartAggJoin
40	2	3	2	PartAggJoin
41	5	1	3	PartAggJoin
42	4	1	3	PartAggJoin
43	3	2	3	PartAggJoin
44	4	1	4	PartAggJoin
45	3	2	3	PartAggJoin

Table A.1: A detailed description of the queries in the Syn-TPC-H dataset. #relations describes the number of involved relations and #aggregates and #grouped the number of aggregate expressions and number of attributes in GROUP BY statements.

APPENDIX B

Queries and Query Plans

The details of the queries q33, q39 and q22 of the Syn-TPC-H dataset.

```
SELECT
    r_name AS Region,
    n_name AS Nation,
    COUNT(DISTINCT s_suppkey) AS Supplier_Count,
    MIN(ps_supplycost) AS Min_Supply_Cost,
    MAX(ps_availqty) AS Max_Available_Quantity
FROM
    region
JOIN
    nation ON r_regionkey = n_regionkey
JOIN
    supplier ON n_nationkey = s_nationkey
JOIN
    partsupp ON s_suppkey = ps_suppkey
GROUP BY
    r_name, n_name
ORDER BY
    Supplier_Count DESC;
```

Figure B.1: q22 of the Syn-TPC-H dataset.

```
SELECT
    c_custkey AS Customer_ID,
    o_orderstatus AS Order_Status,
    AVG(l_extendedprice) AS AVG_Sales
FROM
    customer
JOIN
    nation ON c_nationkey = n_nationkey
JOIN
    region ON n_regionkey = r_regionkey
JOIN
    orders ON c_custkey = o_custkey
JOIN
    lineitem ON o_orderkey = l_orderkey
JOIN
    supplier ON l_suppkey = s_suppkey
GROUP BY
    c_custkey, o_orderstatus;
```

Figure B.2: q33 of the Syn-TPC-H dataset.

```
SELECT
    s_suppkey AS Supplier_ID,
    p_brand AS Part_Brand,
    AVG(l_extendedprice) AS AVG_Sales
FROM
    supplier
JOIN
    nation ON s_nationkey = n_nationkey
JOIN
    partsupp ON s_suppkey = ps_suppkey
JOIN
    part ON ps_partkey = p_partkey
JOIN
    lineitem ON p_partkey = l_partkey
JOIN
    orders ON l_orderkey = o_orderkey
GROUP BY
    s_suppkey, p_brand
ORDER BY
    AVG_Sales DESC;
```

Figure B.3: q39 of the Syn-TPC-H dataset.

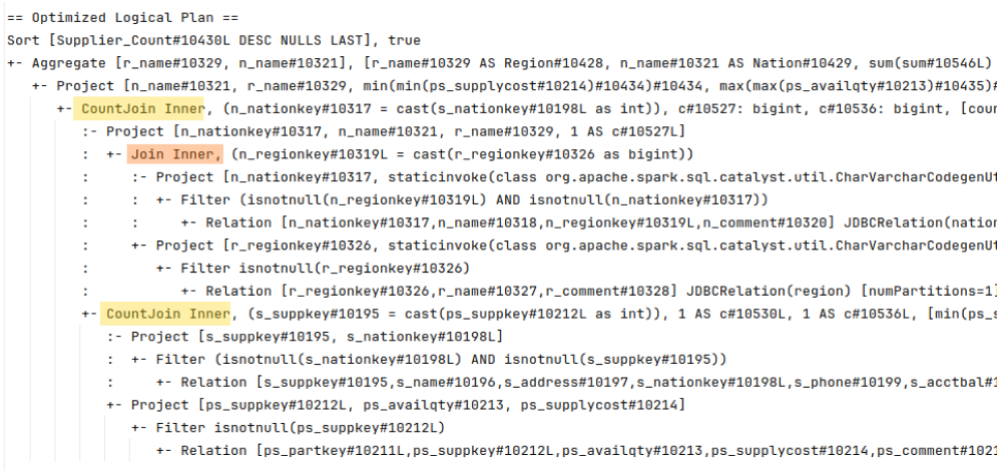


Figure B.4: The logical plan of q22.

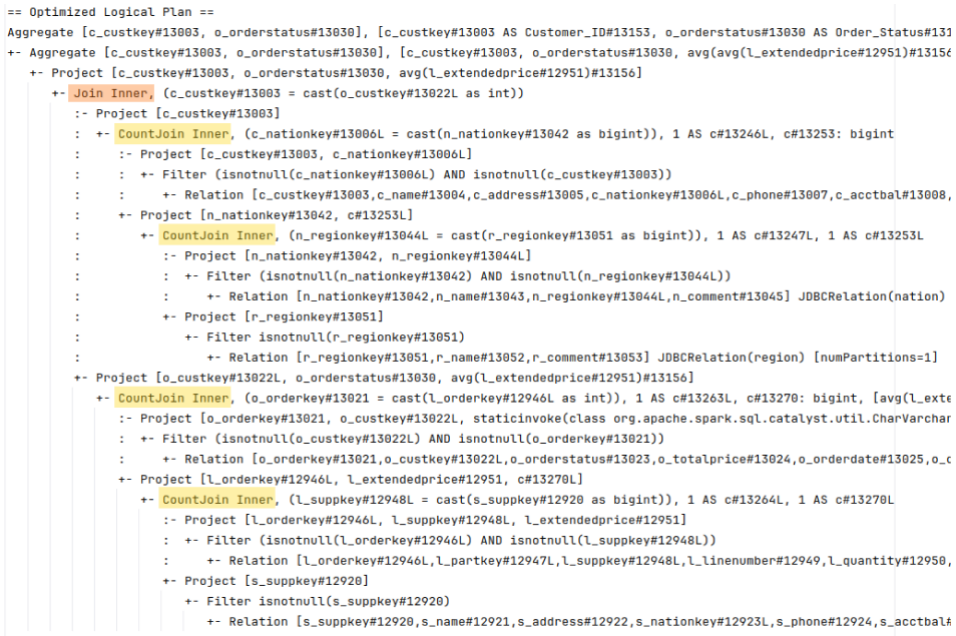
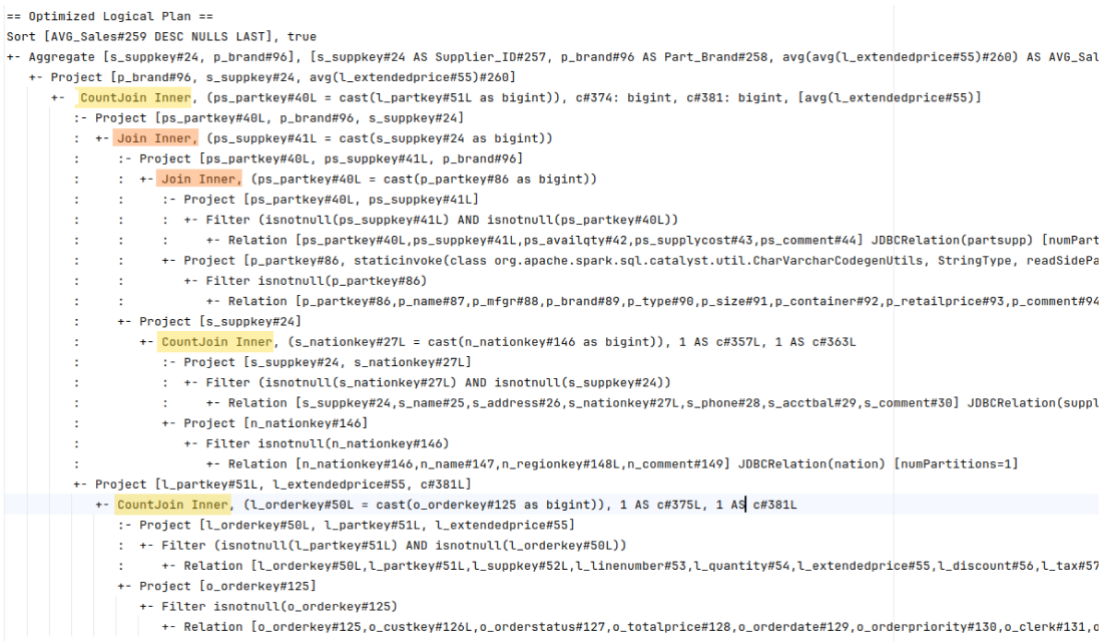


Figure B.5: The logical plan of q33.



```
== Optimized Logical Plan ==
Sort [AVG_Sales#259 DESC NULLS LAST], true
+- Aggregate [s_suppkey#24, p_brand#96], [s_suppkey#24 AS Supplier_ID#257, p_brand#96 AS Part_Brand#258, avg(avg(l_extendedprice#55)#260) AS AVG_Sal
   +- Project [p_brand#96, s_suppkey#24, avg(l_extendedprice#55)#260]
      +- CountJoin Inner, (ps_partkey#40L = cast(l_partkey#51L as bigint)), c#374: bigint, c#381: bigint, [avg(l_extendedprice#55)]
         :- Project [ps_partkey#40L, p_brand#96, s_suppkey#24]
         :  +- Join Inner, (ps_suppkey#41L = cast(s_suppkey#24 as bigint))
         :     :- Project [ps_partkey#40L, ps_suppkey#41L, p_brand#96]
         :     :  +- Join Inner, (ps_partkey#40L = cast(p_partkey#86 as bigint))
         :     :     :- Project [ps_partkey#40L, ps_suppkey#41L]
         :     :     :  +- Filter (isnotnull(ps_suppkey#41L) AND isnotnull(ps_partkey#40L))
         :     :     :     +- Relation [ps_partkey#40L,ps_suppkey#41L,ps_availqty#42,ps_supplycost#43,ps_comment#44] JDBCRelation(partsupp) [numPart
         :     :     +- Project [p_partkey#86, staticinvoke(class org.apache.spark.sql.catalyst.util.CharVarcharCodegenUtils, StringType, readSidePa
         :     :        +- Filter isnotnull(p_partkey#86)
         :     :           +- Relation [p_partkey#86,p_name#87,p_mfgr#88,p_brand#89,p_type#90,p_size#91,p_container#92,p_retailprice#93,p_comment#94
         :     +- Project [s_suppkey#24]
         :        +- CountJoin Inner, (s_nationkey#27L = cast(n_nationkey#146 as bigint)), 1 AS c#357L, 1 AS c#363L
         :           :- Project [s_suppkey#24, s_nationkey#27L]
         :           :  +- Filter (isnotnull(s_nationkey#27L) AND isnotnull(s_suppkey#24))
         :           :     +- Relation [s_suppkey#24,s_name#25,s_address#26,s_nationkey#27L,s_phone#28,s_acctbal#29,s_comment#30] JDBCRelation(suppl
         :           +- Project [n_nationkey#146]
         :              +- Filter isnotnull(n_nationkey#146)
         :                 +- Relation [n_nationkey#146,n_name#147,n_regionkey#148L,n_comment#149] JDBCRelation(nation) [numPartitions=1]
         +- Project [l_partkey#51L, l_extendedprice#55, c#381L]
            +- CountJoin Inner, (l_orderkey#50L = cast(o_orderkey#125 as bigint)), 1 AS c#375L, 1 AS c#381L
               :- Project [l_orderkey#50L, l_partkey#51L, l_extendedprice#55]
               :  +- Filter (isnotnull(l_partkey#51L) AND isnotnull(l_orderkey#50L))
               :     +- Relation [l_orderkey#50L,l_partkey#51L,l_suppkey#52L,l_linenumber#53,l_quantity#54,l_extendedprice#55,l_discount#56,l_tax#57
               +- Project [o_orderkey#125]
                  +- Filter isnotnull(o_orderkey#125)
                     +- Relation [o_orderkey#125,o_custkey#126L,o_orderstatus#127,o_totalprice#128,o_orderdate#129,o_orderpriority#130,o_clerk#131,o
```

Figure B.6: The logical plan of q39.

Overview of Generative AI Tools Used

We used ChatGPT with the models GPT-3.5 and GPT-4 for the following use-cases:

Use-case 1: Formatting Latex code

AI was used to format and re-format latex code such as tables, trees and figures with subfigures.

Example prompt:

"Please arrange the following table as a tree in the following order: Medication is the root, Dog is a child of Medication and Owner and Visit are children of Dog.

[referring to the table in Figure 3.11]

Use-case 2: Data Generation

AI was used to generate a small database of data to refer to throughout the thesis. This data was then edited and expanded by the authors.

The following are some representative prompts that serve as example:

- Can you please generate 6 rows of data for the following table of medication for dogs in latex table row format. one of them should be Bravecto: id, name, brand, dosage, price
- thank you! now please generate 7 entries for the following table of dogs: ID, Name, Breed, Weight, Medication, OwnerID
- thank you! now please generate 4 entries for the table owners as below: ID, Firstname, lastname, Addr, Bank, Tel
- ...

Use-Case 3: Generate Latex Pseudo Code from our Written Pseudo Code

AI was also used to transform our written pseudo-code into latex format Prompt: Please generate latex pseudo code from my written pseudo-code. This is a code of an algorithm. [valid latex pseudo code of another example]

Please format the following pseudo code in the same way in latex as the code above: PartAggJoin: S joins R (R = parent node, S = child node)

input: two lists R,S of tuples with the same values of the join attributes input: List $I_s of indices of aggregate attributes Agg_s, present in both R and S (need to be updated in R) input$: $List I_r of indices of aggregate attributes Agg_r, present only in R -> only the freq. of the tuple that join has to$ $List G_s that contain all applicable grouping attributes in S$

```
Function PartAggJoin:
    if(G_s.isEmpty):
        return AggJoin(R, S, I_s, I_r)
    else:
        if (S.isLeaf):
            // Initialize values
            foreach s in I_s do:
                if A_s in {MIN, MAX} then val_s <- init[s]
                else if A_s in {SUM, COUNT} then val_s <- 0

            // Add attributes c = 1 to every tuple in S
            // Add val_s for every Agg_i in I_s to S
            S.initialize(I_s)

        // Join R and S
        Join(R,S)

        // Project c and Agg_i so that R has the same values as S for c and
        projected_tuples <- project(R, c, Agg_i)


        foreach tuple t in projected_tuples do:
            if t exists in R:
                // propagate value from S to R for Agg_i
                foreach i in I_s do:
                    if A_i is MIN then R[t][i] <- min(R[t][i], S[t][i])
                    else if A_i is MAX then R[t][i] <- max(R[t][i], S[t][i]
                    else if A_i is SUM then R[t][i] <- R[t][i] + S[t][i]
                    else if A_i is COUNT then R[t][i] <- S[t][i] * R[t][i]
```

```
        //propagate c from S to R
        foreach i in I_r do:
            R[t][i] <- R[t][i] * S[t].count

    return R
```

List of Figures

List of Tables

List of Algorithms

Bibliography

[AC99] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 181–192, New York, NY, USA, 1999. Association for Computing Machinery.

[Apa09] Apache spark project documentation, 2009. http://spark.apache.org (accessed: 12.03.2025).

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[BB16] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3), December 2016.

[BKN24] Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proc. VLDB Endow.*, 17(11):3215–3228, July 2024.

[BMT20] A. Bonifati, W. Martens, and T. Timm. An analyitcal study of large sparql query logs. *The VLDB Journal 29*, page 655–679, 2020.

[CHW+22] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Ken Salem. Accurate summary-based cardinality estimation through the lens of cardinality estimation graphs. *Proc. VLDB Endow.*, 15(8):1533–1545, April 2022.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Symposium on the Theory of Computing*, 1977.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[Fag83] Ronald Fagin. Acyclic database schemes (of various degrees): A painless introduction. In Giorgio Ausiello and Marco Protasi, editors, *CAAP'83, Trees in Algebra and Programming, 8th Colloquium, L'Aquila, Italy, March 9-11, 1983, Proceedings*, volume 159 of *Lecture Notes in Computer Science*, pages 65–89. Springer, 1983.

[FGLP21] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics*, 26, July 2021.

[GLL+23a] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice, 2023.

[GLL+23b] Georg Gottlob, Matthias Lanzinger, Davide Mario Longo, Cem Okulmus, Reinhard Pichler, and Alexander Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice, 2023. v2 of the paper.

[GLPN93] Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.

[GLS01] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, May 2001.

[GPS22] Lukas Grasmann, Reinhard Pichler, and Alexander Selzer. Integration of skyline queries into spark sql, 2022.

[Gra79] Marc Graham. On the universal relation (tech. rep.). *University of Toronto*, 1979.

[GTH+20] Fayed F. M. Ghaleb, Azza A. Taha, Maryam Hazman, Mahmoud Abd Ellatif, and Mona Abbass. Rdf-bf-hypergraph representation for relational database. *International Journal of Mathematics and Computer Science*, 15:41–64, 01 2020.

[HW23] Xiao Hu and Qichen Wang. Computing the difference of conjunctive queries efficiently. *Proc. ACM Manag. Data*, 1(2), June 2023.

[HWW+21] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Cardinality estimation in dbms: a comprehensive benchmark evaluation. *Proc. VLDB Endow.*, 15(4):752–765, dec 2021.

[KAK+14] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

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

[LK14] Jure Leskovec and Andrej Krevl. Snap datasets: Stanford large network dataset collection, 2014. http://snap.stanford.edu/data (accessed: 11.09.2024).

[LPS24] Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. Avoiding materialisation for guarded aggregate queries, 2024.

[LPS25] Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. Avoiding materialisation for guarded aggregate queries. Extension of AggJoin to GroupAggJoin, 2025.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 28–36, New York, NY, USA, 1988. Association for Computing Machinery.

[MLK+21] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. Lsqb: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '21, New York, NY, USA, 2021. Association for Computing Machinery.

[Pic23] Reinhard Pichler. Lecture notes in database theory, November 2023.

[Pip21] Antonio Pipita. Dynamic query optimization in spark. Master's thesis, Politecnico Milano, 2021. https://www.politesi.polimi.it/handle/10589/186065 (accessed: 14.04.2025).

[pro] Apache Spark project. Performance tuning. https://spark.apache.org/docs/latest/sql-performance-tuning.html (accessed: 14.04.2025).

[R-p93] R-project for statistical computing. documentation, 1993. http://www.r-project.org (accessed: 12.03.2025).

[SBO13] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for scala, 2013.

[SOAK+19] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. A layered aggregate engine for analytics workloads. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1642–1659, New York, NY, USA, 2019. Association for Computing Machinery.

[TGR22] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Toward responsive dbms: Optimal join algorithms, enumeration, factorization, ranking, and dynamic programming. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3205–3208, 2022.

[TPCa] Tpc-ds benchmark. https://www.tpc.org/tpcds/ (accessed: 30.10.2024).

[TPCb] Tpc-h benchmark. https://www.tpc.org/tpch/ (accessed: 11.09.2024).

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, page 82–94. VLDB Endowment, 1981.

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

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 2, USA, 2012. USENIX Association.