TU WIEN Informatics

Strukturbasierte Abfrageoptimierung in Spalten-Orientierten Datenbanken

zur Erlangung des akademischen Grades

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Jakob Aichinger
Matrikelnummer 11814579

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

Wien, 1. Juli 2024

Jakob Aichinger	Reinhard Pichler

TU WIEN Informatics

Structure-Guided Query Optimization in Column-Stores

submitted in partial fulfillment of the requirements for the degree of

in

Software Engineering and Internet Computing

by

Jakob Aichinger
Registration Number 11814579

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

Vienna, July 1, 2024

Jakob Aichinger	Reinhard Pichler

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

Erklärung zur Verfassung der Arbeit

Jakob Aichinger

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, 1. Juli 2024

Jakob Aichinger

Danksagung

Ich möchte mich bei allen bedanken, die mich während des gesamten Prozesses der Erstellung dieser Arbeit unterstützt und motiviert haben.

In erster Linie bin ich Professor Reinhard Pichler für seine Hilfe sehr dankbar. Von der Unterstützung bei der Themenfindung bis hin zum kontinuierlichen und konstruktiven Feedback war seine Betreuung von großer Bedeutung für mich. Als nächstes möchte ich Alexander Selzer danken, der mich bei vielen technischen Fragen unterstützt hat und dessen Input oft entscheidend für die Bewältigung vieler Herausforderungen war, auf die ich gestoßen bin.

Des Weiteren möchte ich mich bei meinen Eltern, meiner Schwester und meinen Freunden bedanken, die mich die ganze Zeit über motiviert haben.

Zum Schluss möchte ich mich noch bei den Mitarbeitern des Cafe Kafka bedanken, wo ich einen großen Teil der Zeit an dieser Arbeit verbracht habe. Der dort konsumierte Kaffee und die einladende Atmosphäre haben sehr zu meiner Produktivität beigetragen.

Acknowledgements

I would like to thank everyone who supported and motivated me throughout the process of writing this thesis.

First and foremost, I am extremely grateful to Professor Reinhard Pichler for his guidance and support. From helping me identify an interesting topic to providing continuous and constructive feedback, his mentorship has been very important to me. Next, I would also like to thank Alexander Selzer who supported me many times when it came to technical questions and his input was often crucial in overcoming many challenges I encountered.

Furthermore, I would like to thank my parents, my sister, and my friends for giving me motivation throughout this whole time.

Lastly, I would like to thank the staff at Cafe Kafka, where I spent a significant amount of time working on this thesis. Their coffee and welcoming atmosphere greatly contributed to my productivity.

Kurzfassung

In den letzten Jahren hat das Aufkommen von datengesteuerten Bereichen wie Datenwissenschaft, künstliche Intelligenz und Business Intelligence die Nachfrage nach effizienten Datenspeicherlösungen erheblich gesteigert. Infolgedessen sind Datenbankmanagementsysteme (DBMS) von entscheidender Bedeutung geworden, wobei spaltenbasierte Systeme aufgrund ihrer außergewöhnlichen Leistung bei großen, leseintensiven analytischen Arbeitslasten an Beliebtheit gewonnen haben. Eine grundlegende Operation in diesen Systemen ist der Join, bei der Daten aus mehreren Relationen verknüpft werden. Die effiziente Verarbeitung von Join-Abfragen, insbesondere von solchen, die zahlreiche Relationen umfassen, stellt jedoch nach wie vor eine Herausforderung dar, da zu viele und in vielen Fällen unnötige Zwischenergebnisse erzeugt werden. Diese Zwischenergebnisse sind häufig viel größer als die endgültige Ausgabe, was zu einem erheblichen Speicherverbrauch und einer geringeren Leistung führt, insbesondere bei Aggregatabfragen. Während sich spaltenbasierte DBMS in der Regel durch die Ausführung von Aggregatabfragen auszeichnen, kann die Explosion von Zwischenergebnissen während der Abfrageverarbeitung ihre Effizienz stark beeinträchtigen.

Interessanterweise wurde in der jüngsten Forschung eine neuartige Optimierungstechnik für genau dieses Problem entdeckt. Durch die Anwendung einer Teilausführung des so genannten Yannakakis-Algorithmus ist es unter bestimmten Bedingungen möglich, diese unnötigen Zwischenergebnisse zu vermeiden und damit die Leistung dieser Abfragen zu verbessern. Dieser Ansatz unterscheidet sich von herkömmlichen Abfrageoptimierungstechniken, da keine Kardinalitätsschätzungen verwendet werden, sondern der Optimierer bestimmte strukturelle Eigenschaften einer Abfrage nutzt.

Trotz ihres Potenzials wurde diese Optimierungstechnik bisher noch nicht in ein spaltenbasiertes Datenbanksystem integriert. Diese Arbeit zielt darauf ab, diese Lücke zu schließen, indem diese Optimierungstechnik in ClickHouse integriert wird, das laut DB-Engines als der derzeit beliebteste Spaltenspeicher angesehen werden kann. Die Ergebnisse sind sehr vielversprechend und zeigen, dass Abfragen, die normalerweise zu einer Zeitüberschreitung führen würden, dank dieser Optimierung nun effizient und ohne Probleme ausgeführt werden können.

Abstract

In recent years, the rise of data-driven fields such as data science, artificial intelligence, and business intelligence has significantly increased the demand for efficient data storage solutions. As a result, database management systems (DBMS) have become crucial, with column-based systems gaining popularity for their exceptional performance in large-scale, read-heavy analytical workloads. A fundamental operation in these systems is the join, which combines data from multiple relations. However, efficiently processing join queries, especially those involving numerous relations, remains challenging due to the generation of excessive, and in many cases unnecessary, intermediate results. These intermediate results are frequently much larger than the final output, leading to significant memory usage and reduced performance, particularly in the case of aggregate queries. While column-stores typically excel in executing aggregate queries, the explosion of intermediate results during query processing can severely undermine their efficiency.

Interestingly, recent research discovered a novel optimization technique for exactly this problem. By applying a partial execution of the so-called Yannakakis' algorithm, it is possible under certain conditions to avoid producing these unnecessary intermediate results and thereby improve the performance of these queries. This approach is different from traditional query optimization techniques, as no cardinality estimates are used, but instead, the optimizer uses certain structural properties of the query.

Despite its potential, this optimization technique has yet to be integrated into any column-based database system. The implementation is particularly challenging due to the impedance mismatch with the Volcano Query Evaluation Model, which is commonly used by many DBMS. This thesis aims to fill that gap by integrating this optimization technique into ClickHouse, which can be considered the most popular column-store at the moment according to the rankings from DB-Engines. The results are highly promising and show that queries that would typically timeout can now be executed efficiently without issues, thanks to this optimization.

Contents

CHAPTER 1

Introduction

1.1 Background and Motivation

In recent years, the emergence of data-driven fields, such as data science, artificial intelligence, and business intelligence, has greatly impacted various industries and gained huge popularity. As a result, efficient data storage has become more critical than ever before, and database management systems (DBMS) are playing an increasingly important role. The choice of using a certain DBMS often depends on the workload requirements. In the case of the analytics field, column-based database systems have become particularly popular due to their good performance in handling large-scale, read-heavy operations [21].

One of the essential features of database systems is the ability to join different sources of data (relations) together. This is especially relevant in business intelligence, where tools are used for automatically generating queries with multiple join operations, sometimes involving several hundred relations [37]. Although the join is one of the most fundamental operations, if not the most fundamental aspect of query evaluation, efficiently evaluating join queries still remains a challenging task for today's DBMS.

An approach for optimizing join query performance is determining the most efficient order in which relations are joined. The sequence of join operations can significantly impact the query's performance by influencing the size of the intermediate results produced during execution. In some cases, optimizing the join order can substantially reduce these intermediate results, leading to more efficient query processing. However, even with such optimizations, the problem known as intermediate result explosion, where the intermediate data grows excessively large, continues to be a critical bottleneck in query execution [30]. This issue is especially common in complex queries that involve many joins, where the size of intermediate results can quickly become unmanageable and lead to significant performance degradation. In the case of aggregate queries, where

column-stores are typically favored for their performance benefits, this challenge becomes even more significant. Since only a small portion of the join result is ultimately needed, avoiding the materialization of the entire result set is highly desirable.

However, if we restrict the problem to Acyclic Conjunctive Queries (ACQs) a solution to the issue of producing unnecessary intermediate results has been found for a long time. The so-called Yannakakis' algorithm [44] allows under specific circumstances to compute a join query in polynomial time of input and output size [29] and therefore makes the computation of a join query optimal. This approach makes use of the structural properties of an acyclic conjunctive query. The studies of the Yannakakis' algorithm show that for these type of queries, it is indeed possible to completely eliminate the problem of producing unnecessary intermediate results during query evaluation [44].

Furthermore, recent research [31] has discovered that a specific class of queries, known as zero-materialization aggregate (0MA) queries, can be answered even with a partial execution of Yannakakis' algorithm. This partial execution of Yannakakis' algorithm involves only using semi-joins and therefore not only solves the issue of producing unnecessary intermediate results, but in fact eliminates the need for materialising join results.

1.2 Research Problem and Objectives

While from a theoretical standpoint the approach of using Yannakakis' algorithm is highly promising, practically to the best of our knowledge no such implementation has been done in the case of a state-of-the-art column store database system.

To close this gap between theory and practice, the objective of this thesis is to implement the first phase of Yannakakis' algorithm, specifically the bottom-up semi-join, into ClickHouse [3], which can be considered the most popular column-store at the moment according to [11]. This integration is non-trivial and can present challenges, particularly in understanding how ClickHouse translates queries into internal data structures and processes them. Additionally, there is a potential impedance mismatch between the way the optimizer operates and ClickHouse's internal query execution, which can complicate the integration. More details on this can be found in Section 4.2. The primary research questions we aim to answer is whether such an integration is first of all possible and secondly investigate the potential performance gains that come from using this optimization technique.

1.3 Methodology

In order to achieve the objectives mentioned in the previous section, the following steps will be taken.

- **Implementation of a Structure-Guided query optimizer**

The implementation phase begins with an in-depth examination of ClickHouse's architecture, focusing specifically on query parsing and processing mechanisms. Following this, the first part of Yannakakis' algorithm will be integrated. This involves constructing a join tree using GYO reduction [44] and ensuring that the algorithm seamlessly fits within ClickHouse's query processing workflow.

- **Data Acquisition and Evaluation**

  Selecting an appropriate dataset is critical for the evaluation phase. The dataset must be substantial in terms of the number of rows and tables to adequately test the system's performance. Additionally, a set of queries will be needed for the evaluation. Based on [30], where the effectiveness of Yannakakis' algorithm was tested on the MusicBrainz dataset, a similar setup is used in this thesis.

  The evaluation will compare the performance of the original ClickHouse implementation against the modified version using the structure-guided query optimizer. Key metrics such as query execution time and memory consumption will be measured for all tested queries. After benchmarking, the results will be analyzed and interpreted to understand the performance differences, identifying reasons for any observed increases or decreases in efficiency compared to the baseline (the original ClickHouse system).

1.4 Organization of the Thesis

The first two Chapters, 2 and 3, can be seen as preliminary and introduce the background of this thesis.

Chapter 2 starts by exploring column-stores. It looks into the storage model, key features, influential systems, and practical applications of column-stores, with a specific focus on ClickHouse, which is the system used for the practical part.

Chapter 3 discusses query optimization in general, the existing problems, particularly with join queries, and finally, how a new approach, structure-guided query optimization, can potentially solve the issue of intermediate result explosion.

Following this, Chapter 4 covers the practical part of this thesis, which involves integrating a structure-guided query optimizer into the open-source column-store ClickHouse. It discusses query processing in ClickHouse, the challenges and considerations, the integration process, and other modifications made to ClickHouse.

Chapter 5 explains the setup and methodology used to test the system, describes the test data, queries, and hardware, outlines the performance metrics and evaluation criteria, and presents and analyzes the results obtained.

Finally, the conclusion summarizes the findings of the research and suggests possible directions for future work.

CHAPTER 2

Column-Stores

In this chapter, it will be explained what *column-oriented database systems*, often also called *column-stores*, are and why they are used. Since the main part of this thesis lies in query optimization for one specific column-store system (ClickHouse), understanding the architecture and benefits of using such a system is essential.

In Section 2.1, the data layout (storage model) of column-oriented and row-oriented databases will be compared. Section 2.2 talks about important characteristics of column-stores and what optimization techniques are used specifically for these types of database systems. Section 2.3 introduces three important column-store systems that have a significant influence on the typical features and optimization techniques that can be found in modern column-stores. The last Section 2.5 talks about typical use cases for column-store and use cases that have been discovered where they have shown to be advantageous compared to the row-store architecture.

2.1 Storage Model

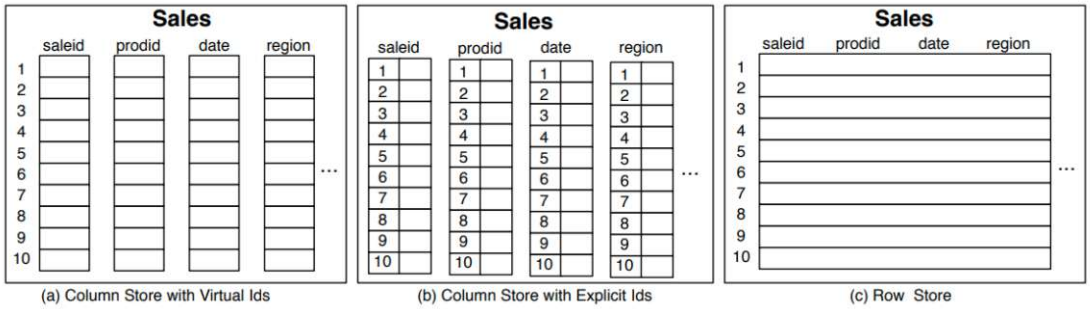


Figure 2.1: Physical layout of column-oriented and row-oriented databases [21]

In Figure 2.1, the main difference regarding the storage model of column-oriented and row-oriented database systems is illustrated. For both types, a table *Sales* with various attributes is stored. The main difference lies in the fact that for the row-oriented approach (2.1c), data is stored row-wise, and in the column-oriented approach (2.1a,b), data is stored in columns.

In practice, this means that when a user wants to access a single attribute e.g. *prodid* from table *Sales*, the column-store system only needs to access this specific entry from the corresponding column. In contrast, the row store would need to read the entire row from disk and discard all attributes that the user does not request. In this particular use case, one can already see that column stores have a significant advantage when it comes to requests that read multiple entries from a single column because the layout naturally allows the system to do this efficiently. These requests are especially prominent in the analytical world, where operations involving aggregate functions are frequently used. The main downside is that as soon as a user needs more than one column from a table, the column-store needs to perform multiple searches for all the requested columns, which is not the case for row-oriented databases.

2.2 Key Features

It was already mentioned that column-stores are known to be very efficient and fast in query processing for certain use cases. In [23], key features that substantially contribute to this efficiency gain in query processing are analyzed. In this section, the features of column-stores will be discussed and explained.

2.2.1 Virtual and Explicit Ids

One problem with storing data in columns is that the database system somehow needs to keep track of how different columns relate to each other. This is, for example, important when a user wants to access multiple attributes from one tuple, e.g., access the *region* for one particular *saleid*. Similarly to the relational model where ids are used to store information on how different **tables** relate to each other, column-stores use ids to store information on how different **columns** relate to each other.

Essentially, there are two different forms of how such a relation is represented, **virtual Ids** and **explicit Ids**, which can also be seen in Figure 2.1a,b. **Explicit ids** are very simple and will give every attribute of a tuple a specific id. Later, if one wants to access a particular tuple, this id is used as a unique identifier that indicates that specific entries from different columns belong together. The biggest downside is that first storing this identifier for every attribute introduces a lot of overhead and is not very efficient in terms of storage. Also, performing an equality check on the Ids to find a specific tuple is similar to performing a join operation and, therefore, very costly. An alternative way

that mitigates these issues is using **virtual Ids**. Here, the database system stores each column in a fixed-size vector. Records will then be stored at the same position over multiple columns so that the position of the record introduces a virtual id. This makes it easy for a column-store to retrieve multiple column attributes from a single record. Although virtual Ids can solve the problem of introducing storage overhead and efficiently finding columns that belong together, we will later see when we talk about compression that fixed-size vectors can also have certain disadvantages.

2.2.2 Compression

Compression is an essential concept for data storage and works exceptionally well when the data has high data value locality [23], meaning that similar data is stored together. Different types of data can yield different compression ratios. For row-oriented systems, precisely this can be a problem since the tuple that needs to be compressed can often contain columns with different data types e.g., *date*, *region*, and *phone number*, making compression algorithms much less efficient.

Column-stores, on the other hand, don't have to deal with this problem since data is stored column-wise, which allows the system to store data with the same data types together. If the column is then also sorted, this can lead to very high homogeneity of data, and compression algorithms will yield excellent results.

An obvious improvement that comes from the fact that compression algorithms work so well for column-stores is that disk space can heavily be reduced. However, compression is not only important for saving disk space but can also have a big influence on performance. The reason is that data is usually read from disk into memory, and compression allows the system to reduce the time spent on doing so, leading to better I/O performance and making query processing faster.

Various compression algorithms are used in column-stores including run-length encoding, bit-vector encoding, and dictionary compression. Run-length encoding (RLE) is straightforward and replaces a "run" of the same value (e.g., 1,1,1,1) by a triplet (value, start position, run-length) (e.g., 1, 1, 4). In this example, it can be seen that decompression is not necessarily needed since operations such as computing the sum can directly be done on the compressed data. While applying compression techniques such as RLE can drastically improve I/O performance, particularly when the target column contains many runs, it has the disadvantage of introducing complexity to the process of tuple reconstruction across multiple columns, as it is not feasible to generate virtual IDs using a fixed-size column. It is also important to mention that using compression introduces additional complexity when it comes to inserting and updating existing data.

2.2.3 Late Materialization

In many use cases, queries would read not only a single attribute from one entity but multiple ones, and thus, the system needs to access the corresponding columns for all attributes. Since these columns are stored separately in column-stores, a very frequent operation that needs to be performed is linking different columns together in order later to present them in the form of a tuple. It was already said in Section 2.2.1 that this can be done using ids, and linking different columns together is similar to a join operation.

Naive column stores would link multiple columns, also called materialization, at the beginning and then perform operations on these constructed tuple rows. This type of processing is referred to as "early materialization". It has the disadvantage that the full potential of the column-store data model is not used in performance optimization.

Modern column-stores, on the other hand, make use of a "late materialization," meaning that data is kept in columns as long as possible, and certain operations, such as filtering, will directly be performed on a specific column. An example of how such late materialization can be achieved is shown in 2.2. Here, two relations R and S and a particular query are given. In (1), one can see that a filtering operation is directly applied on the column $R.a$. The result is a position list *inter1* containing a list of indices of qualifying tuples. In the second step (2), this list *inter1* is used to select the corresponding values of column $R.b$, which is then again filtered on the second predicate in step 3.

Since these positional lists can be represented as a bit string, a bit-wise AND can be used to compute the intersection of two filter operations, allowing the system to perform these kinds of operations in a very efficient way.

2.2.4 Block iteration

Rather than individual rows, column-stores both access and process data in blocks. This contradicts the so-called tuple-at-a-time processing, which many row-store systems such as MySQL use [23]. This tuple-at-a-time processing approach, also often called the "Volcano-style" iterator model, incrementally processes individual tuples through a series of operations, maintaining a limited set of intermediate results but at the cost of increased function call overheads. One bottleneck with this is that for every tuple that is processed, the needed attributes need to be first extracted from the tuple, which leads to the mentioned function call overhead [23].

In column-stores, on the other hand, this is not the case since a batch-oriented approach is used. Blocks of values (vectors) are directly accessed in a single call, and the corresponding values can then be processed as an array. This processing and accessing of data in blocks has many benefits:

1. **Enhanced I/O efficiency:** Reading data in blocks minimizes disk seeks and I/O operations, which is beneficial for large-scale analytical queries.

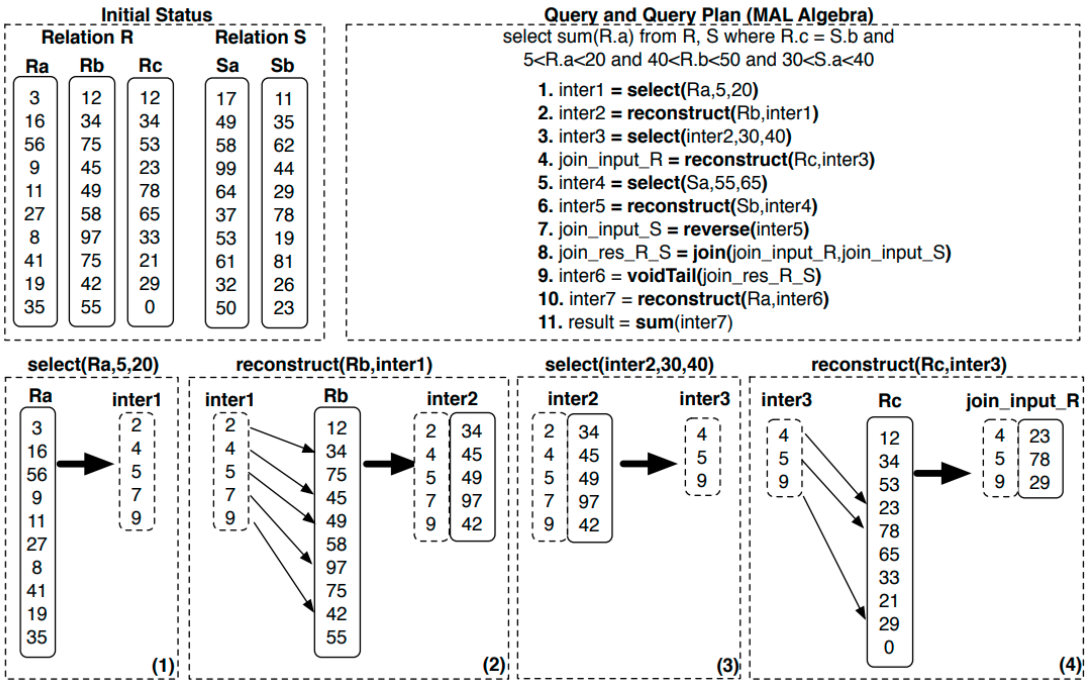


Figure 2.2: Late Materialization [21]

2. **Tailored cache utilization:** Systems such as VectorWise will choose the vector size adequately so that it fits in the CPU cache, leading to better utilization.

3. **Concurrent data handling:** The block iteration model offers great possibilities for parallel processing, which makes it very suitable for modern multi-core processors.

4. **No overhead of unused columns:** In row-stores, all columns of a table are read into memory, even the ones that are not part of the query. This is especially critical if the table contains a large number of columns, but only a small subset of these will be frequently accessed.

2.2.5 Joins

The column oriented data model offers various opportunities for improving the efficiency of join operations. In Section 2.2.3, the concept of *early materialization* and *late materialization* was discussed. For the early materialization strategy, the join operator deals with fully constructed tuples work, therefore similar to row-stores.

For late materialization, on the other hand, there exist several techniques to optimize the join operation. One improvement that is made for joins in column stores with late

materialization is to perform the join only on the columns that are part of the predicate of the join. In the case of a hash join, this can drastically reduce the size of the data being processed and improve access patterns.

In Figure 2.3, one can see how two columns are joined together, resulting in two position lists of the corresponding matches in these columns, e.g., there is a match on position 1 and position 2 with value 42. An important finding is that the left positional list is sorted, while the right one is not. This is because during the join, the left column will be iterated in order, and at the same time, the right column will then be checked for matches via, e.g., a hash table. Unsorted positional output like this can be problematic

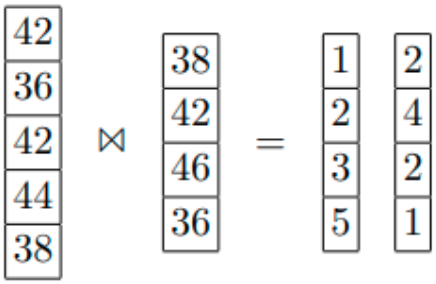


Figure 2.3: [21]

for future operations performed on this data and is a major bottleneck. The reason is that with late materialization, we only work with a single column (the join predicate). We later need to construct the tuple, and for this, the position list should ideally be in order. In the research community, a number of approaches exist to solve this problem, with one of them being the so-called *Jive Join*.

The basic idea of a Jive join is to introduce an additional column of increasing integers to optimize the extraction of values based on a list of unordered positions. With this additional column, the algorithm aims to enable sequential iteration through all columns. A good example of how this is done on a particular instance can be found in [21].

Due to introducing a lot of extra complexity, late materialization for joins is often avoided, and commercial column-stores use a technique called *hybrid materialization*. The idea is that for the right side of the join, a materialization will be performed in advance, meaning that not only the join predicate is used but all columns relevant to the query. The left side will send only the join predicate column, but since the position list of this side is sorted, the materialization can be done efficiently at a later time.

2.2.6 Indexing

Although data retrieval is very fast in column-stores there is still room for optimization. One way such optimization can be achieved is by using proper indexing. In row-stores,

indexing is typically done by using a specific data structure such as B(+)-tree, which is kept in-memory [26].

For column-stores it has been discovered that while using tree structures provides rapid random access, sorting columns by certain attributes proves to be more beneficial [21]. One way this can be achieved is the idea of projections, which is, for example, used in the column-store system C-Store. Here, tables will be replicated and sorted by different attributes. Although this approach needs additional disk space since duplicate data will be stored, the fact that compression works very well for column-stores makes this overhead not so significant.

An alternative approach that is commonly used in the field of column-stores is the use of so-called **database cracking**, which is a form of **adaptive indexing**. The idea is that instead of deciding upfront which indexes to create, the system dynamically creates an index. This is done by using information from query predicates and then, based on this, creating indexes as needed. A big advantage of this approach is that one does not need to spend time analyzing the expected type of queries and how to optimally create indexes for those in advance. Also, since indexes are created dynamically, it will automatically detect changes in the workload and adapt accordingly.

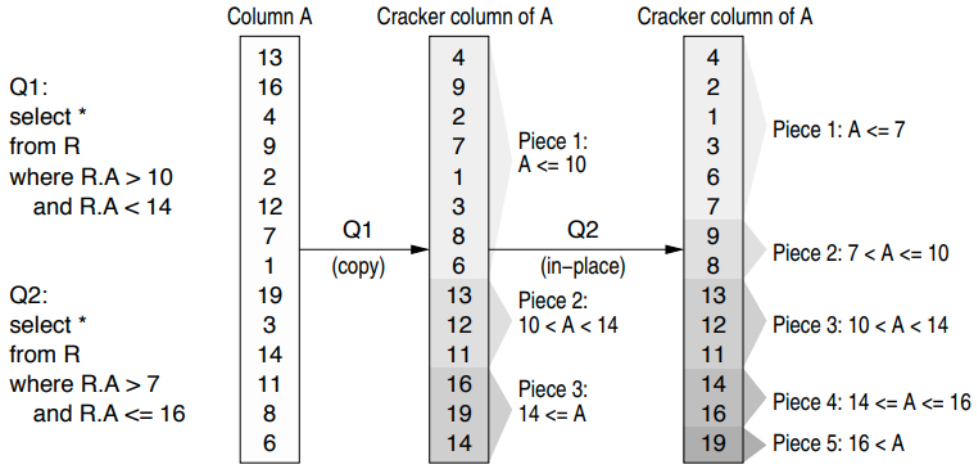


Figure 2.4: Database cracking [21]

The term **database cracking** comes from how column-stores partition (crack) the physical data store depending on the predicates of incoming queries. An example of how this type of adaptive indexing is done can be seen in Figure 2.4 . The first query *Q1* with the two predicates $R.A > 10$ and $R.A < 14$ cracks column A into three pieces accordingly. The second query will further partition the column into multiple pieces.

Another technique that is also often used in both row-stores and column-stores is bitmap indexing. For column-stores, bitmap indexing works particularly well when the column contains only a limited set of values (low cardinality), an example of this could be a column that contains the gender of a person. Here, a bitmap index would map the column values to a series of bits, allowing the system to do operations such as filtering extremely efficiently.

2.3 Influential Systems

In this section, the three column-store systems MonetDB [33], C-Store [41], and Vectorwise [46] are discussed regarding their architecture and important design choices. These systems started as research prototypes and pioneered the general idea of the column store storage model. Several of the key features that were covered in the previous Section 2.2 were initially introduced as part of those systems architectures and have since been adopted by commercial solutions.

2.3.1 MonetDB

MonetDB [33] is an open-source column-store system that has been developed at the CWI (Centrum Wiskunde & Informatica) research center in the Netherlands since 1993. The system plays an essential role in academic research, particularly in the field of column-stores, as in the past two decades, many innovations were first implemented in MonetDB and later adopted by many other systems. It is also worth mentioning that MonetDB is one of the first publicly available column-stores.

Data in MonetDB is always organized in columns, both in the case of persisted data on disk and when it is processed in memory. It is mainly focused on read-oriented workloads. Uptades in MonetDB are handled via a pending update column. Every update operation is first translated into an append operation. Read operations are then done on both the pending update column and the actual column. Periodically, the pending updates will then be merged into the actual column. Differing from many other column-stores, it also supports the use of transactions. Unlike the conventional locking mechanism, which is commonly used in row-stores, MonetDB provides additional tables where the transactions are applied. For indexing, it applies the idea of **database cracking**, which was already discussed in Section 2.2.6.

The main format that is used to both store and process data is called BAT (=Binary Association Table). A BAT always consists of two columns. The first column is called the head and can either be an OID (object-identifier) or the surrogate. The second column (tail) holds the data value of the stored attribute. For one tuple in a dataset, MonetDB

will automatically use the same OID for all the tuple attributes, similar to the concept of explicit indexing, which was discussed in 2.2.1. These BATs are also used through the entire process of query evaluation. Therefore, MonetDB also applies the concept of **late materialization**, meaning that tuples will only be constructed right before they are sent to the client.

The architecture consists of three parts, namely, frontend, backend, and kernel. The frontend can consume different types of queries: SQL, OQL, XQuery, SPARQL, and it will be translated into BAT algebra, which the backend will then process. Internally, MonetDB applies its own language called MAL (=MonetDB Assembly Language). This MAL is heavily used in the backend of MonetDB where, for example, the MAL optimizer lies, which consists of many MAL programs that apply different query optimization.

2.3.2 C-Store

C-Store is a column-store system that was developed by a team of researchers from different universities, namely, Brown University, Brandeis University, Massachusetts Institute of Technology, and the University of Massachusetts Boston.

In C-Store, data is organized into a read-optimized store (ROS) and a write-optimized store (WOS). The ROS consists of column files with compressed data, which is sorted by specific attributes. WOS, on the other hand, holds newly loaded data in an uncompressed format to facilitate efficient data loading. Periodically, data is moved from the WOS to the ROS. This is done through a background process called "tuple mover".

When a query is executed in C-Store, the data will be retrieved from the ROS and WOS. The result from both tables is unified and returned. Regarding optimization techniques for query processing, C-Store also applies the concept of late materialization that was already discussed in Section 2.2.

Another aspect for optimizing query performance in C-Store is using "projections". The idea is to store copies of columns and sort them in different ways so that for most queries, there already exists a projection containing all the attributes, ideally in a sort order that benefits the evaluation of the query. In 2.5, we see an example of this, where two projections for the table 'Sales' are given. They contain different attributes of the original table and have different sort orders, while 2.5 (a) is sorted by date only, 2.5 (b) is sorted by region and date. The idea is that with projection (a), one could easily answer queries that, as an example, would get the number of sales for a specific time frame. If one wants to additionally filter by a specific region, there might already exist a projection similar to (b) that can help to improve the query performance.

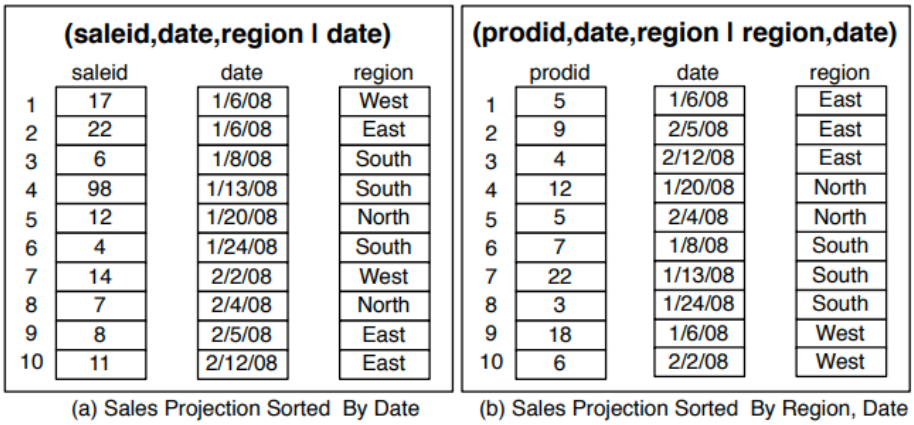


Figure 2.5: Two projections of a table 'Sales' [21]

2.3.3 Vectorwise

The project of Vectorwise [46] was originally initiated in 2003 at the CWI, which is the same institution that started MonetDB. At that time, the project was called X100 and introduced innovative features such as vectorized query processing. As the performance tests of X100 showed promising results, a company called Actian Corp. wanted to make it commercially available. For this, X100 was bought up and integrated into their own DBMS called Ingres, which resulted in Vectorwise. The original developers of X100 wrote the paper [25], which talks about some of the ideas that initiated their project in the paper. This paper later won the "VLDB Test of Time award" [18], which selects a paper from the past that had a significant impact in practice, highlighting the significant influence Vectorwise had on database systems in general and specifically on column stores.

The architecture of Vectorwise consists of some features that were reused from the Ingres DBMS, including connectivity APIs, a Query parser, and a cost-based query optimization based on histograms, and on the other hand, features from X100 that are in the area of query execution and the storage layer. The main innovation that came from X100 and addressed some of the problems in MonetDB was, on the one hand, an optimized approach for **data storage** and, on the other hand, efficient query execution using the so-called **vectorized execution model**.

For **data storage**, Vectorwise employs a flexible data organization model called PAX (Partition Attributes Across). PAX stores tables in so-called PAX partitions. Each partition will then contain a group of columns. How these grouping of columns is applied can be either defined explicitly or otherwise will be done automatically by the system. One notable innovation in this context is that Vectorwise uses additional boolean columns to represent null values, allowing the system to skip null checks during query execution. These special columns that are used for representing null values will always be within the same PAX partition as the column in the original column holding the actual values.

In contrast to regular row-based processing, Vectorwise applies the so-called **vectorized execution model**. Here, the processing of data happens in batches of vectors, typically involving multiple columns at the same time. This method greatly improves performance, especially compared to other column-stores such as MonetDB, since the vectorized processing aligns with the columnar storage format and can apply various optimization techniques. One such optimization technique would be improved CPU cache utilization, thus minimizing the need to fetch data from RAM.

2.4 ClickHouse

This section gives an overview of the open-source column-oriented Database Management System (DBMS) ClickHouse, which is also the system used for the practical part of this thesis in Chapter 4. As of the present day, ClickHouse can be considered one of the most popular column-oriented DBMS. According to the latest ranking by db-engines.com, ClickHouse holds a position at 37 over all types of DBMS [11]. In contrast, competing column-oriented systems such as MonetDB lag significantly behind, currently positioned at rank 148. Although, its substantial increase in popularity in the past years, the amount of academic research related to ClickHouse is limited. There can be found several papers that compare traditional DBMS to ClickHouse in terms of performance, such as against Oracle DBMS [34], MySQL [43], and one comparison that was performed at CERN where it was evaluated against InfluxDB [42]. However, the focus of these papers lies mostly in performance evaluations and lacks in giving insights into why ClickHouse is faster and what it does differently in terms of architecture. Moreover, to our knowledge as of today, there does not exist any research that looks into query optimization of ClickHouse.

In the following, we first look into the historical development of ClickHouse and its roots. Following this, an overview of the key features of ClickHouse is given. The main source used for the key features is the Clickhouse documentation [5]. This should give some insights into some key features of ClickHouse and its historical background.

2.4.1 History

ClickHouse was initially developed at Yandex, with the goal to be used for real time analytics processing at Yandex Metrica, which is the web analytics platform of Yandex for trackinAcquisitionrting website traffic [8]. The initial development started in 2008 by Alexey Milovidov and a small team of developers, and three years later they put it in production at Yandex Metrica and later on used ClickHouse in many other services of the company.

After the great success within Yandex, the developers recognized ClickHouse broad applicability and potential and decided to release it as an open-source project in 2016 under the Apache 2 license. Since its open-source release, ClickHouse has today over one thousand developers contributing to the codebase and is used as part of the infrastructure in many companies worldwide, including Uber's log analytics platform, GitLab's APM

datastore, and Cloudflare's HTTP analytics infrastructure [10], [7], [4]. ClickHouse was also used at CERN, as part of their so-called Trigger and Data Acquisition (TDAQ) system [42].

In 2021 the original developers announced that they moved away from Yandex and founded ClickHouse Inc. The company is headquartered in the San Francisco Bay area. As of today, they have raised $250M Series B at a $2B valuation and the company continues developing the open-source database system under the original lead engineer Alexey Milovidov as CTO.

2.4.2 Key Features

Query execution performance was according to developers the main objective when designing ClickHouse [19]. The increase in its popularity, its adoption by numerous companies, and the results from various performance evaluations demonstrate the achievement of this objective.

One such performance evaluation can be seen in Figure 2.6, where ClickHouse manages to outperform both modern column-oriented systems including DuckDB, and also older systems such as MonetDB. The graphic was produced using the ClickBench benchmark, which allows one to compare the performance of various databases and uses typical queries in the analytics space. It must be considered that the benchmark was made by the creators of ClickHouse, and therefore there can be potential bias. Nevertheless, the results are quite impressive, and there is also a good discussion regarding the comparability of this benchmark, where the database author of Apache Druid offers good input on this topic [15].

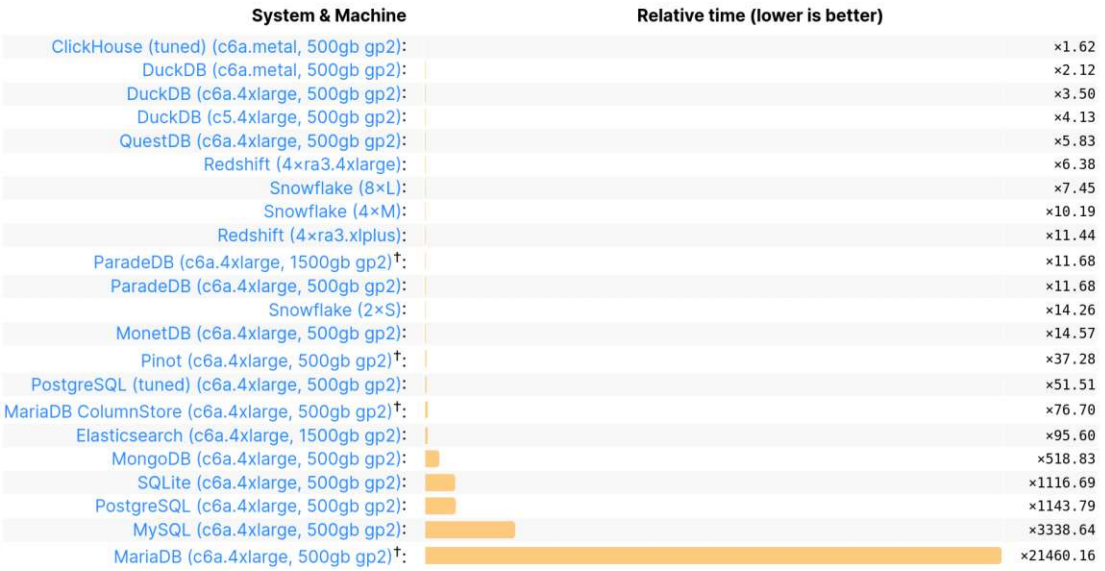


Figure 2.6: ClickBench Sample Benchmark [2]

This raises the question of what exactly ClickHouse does differently than other conventional OLAP systems and where these notable performance capabilities come from.

During a presentation at the Big Data Technology Conference in 2019, this question was explained in detail by the CTO of Clickhouse. In the following, some key features and architectural choices, especially on how data is stored and processed, will be discussed.

Table Engines

ClickHouse employs the concept of so-called table engines [9]. A table engine is a type of table and has implications for many internal aspects such as how and where data is stored. Since ClickHouse was initially built with a focus on filtering and aggregating data as fast as possible for Yandex Metrica, table engines should provide a way of optimizing the database for different use cases. As of today, Clickhouse supports over 20 engines in four categories.

Every time a new table is created in ClickHouse, the table engine must be specified as part of the query which can be seen in the following example query:

```
CREATE TABLE [IF NOT EXISTS] [db.]table_name [ON CLUSTER cluster]
(
    ...
) ENGINE = engine
```

Although there do exist many different table engines including special engines for communicating with other data storage such as the PostgreSQL table engine or special engines for logs, the most prominent one is the **MergeTree engine**, which is also considered the default table engine.

One of the main features of the MergeTree engine is that the stored data is automatically sorted by the primary key and allows the creation of a small sparse index, meaning that not every single row has its own index, more on this will be detailed later. The name MergeTree comes from the fact that each insert creates one or multiple parts for the data, which are regularly merged in the background. This will be explained in 2.4.2.

Data Storage and Indexes

Tables with the engine type MergeTree are stored in so-called *data parts*. Data parts can be stored in one of the two following formats:

- **Wide**: Each column is stored in a separate file.
- **Compact**: All columns are stored in one file.

Each *data part* is further subdivided into *granules*, which are the smallest data sets that ClickHouse reads. A granule holds a specific number of rows, with the default being 8192 rows per granule.

Contrary, to other RDBMS, ClickHouse does not store an index entry for every single row, but instead applies the concept of a sparse index where every granule would have its index.

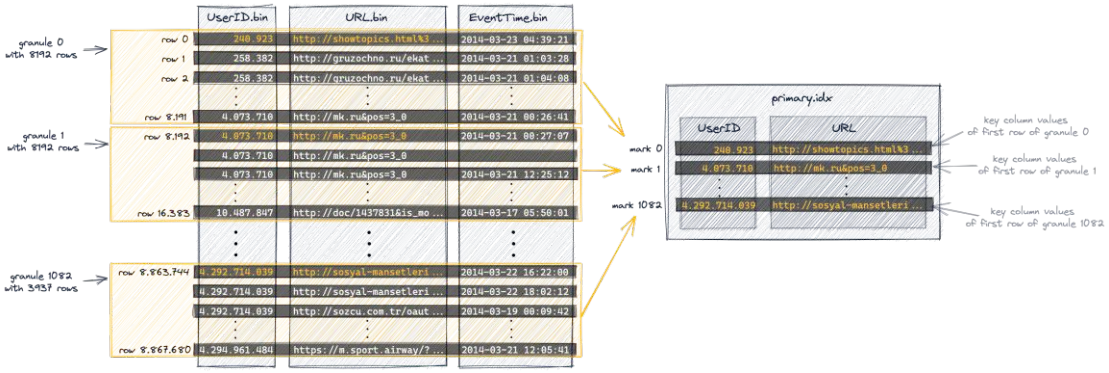


Figure 2.7: Sparse Primary Index Example

In Figure 2.7 one can see an illustration of how ClickHouse would practically store data and apply this concept of sparse indexing. The data contains three different columns `UserID`, `URL`, and `EventTime`. The columns are all stored in different files (*.bin), meaning the wide format is used.

The rows of each file are grouped into granules. In the case of Figure 2.7 the file is organized from granule 0 to granule 1082, with each granule except the last one containing the default value of 8192 rows. Based on these granules, the primary index file is created, as shown on the right in Figure 2.7. This file, named `primary.idx`, is an uncompressed flat array that stores numerical index marks starting at 0. These index marks correspond to the primary key column values for the first row of each granule. For instance, `mark 0` stores the key column values of the first row of granule 0, `mark 1` stores the values of the first row of granule 1, and so forth. The total number of entries in the primary index is 1083, matching the number of granules in the table.

It's important to note that the primary index file is loaded entirely into memory. If the file size exceeds the available memory space, ClickHouse will raise an error. Each index entry serves as a marker for the beginning of a specific data range.

For instance, the `UserID` values in the primary index are sorted in ascending order. Therefore, `mark 1` indicates that `UserID` values in granule 1 and subsequent granules are greater than or equal to 4,073,710. This ordering allows for the application of binary search algorithms for query filtering on the first column of the primary key.

Focus on Low-Level Details

The developers claim that one aspect that makes ClickHouse stand out in terms of performance is their attention to low-level details, specifically when it comes to adapting standard algorithms to different scenarios.

This is demonstrated in many cases, with one being that they tend to avoid built-in algorithms, but instead favor dynamically tailored algorithms depending on the workload. For instance, when it comes to building a hash table in the case of a GROUP BY clause, ClickHouse chooses one of over 30 different variations for each specific query.

Similarly, they looked into many different algorithms for string processing, where they in the end implemented an adapted version of Volnitsky's algorithm, and for sorting ClickHouse uses adapted versions of pdqsort and radix sort [20].

Data Compression

Another factor for achieving high performance according to the developers is data compression. ClickHouse offers both general-purpose compression codecs and specialized codecs for specific types of data. For instance, the DoubleDelta codec calculates deltas of deltas, optimizing compression for monotonic sequences like time series data. Similarly, the Gorilla codec employs XOR operations between consecutive floating-point values to achieve compact binary representation.

Distributed Processing

Column-based systems, including those mentioned in previous sections, typically do not natively support query processing over multiple servers without additional configurations or extensions.

One of the standout features of ClickHouse is its built-in capability to handle distributed query processing through a dedicated table engine.

This feature offers significant advantages for scaling, enabling both the sharding and replication of database tables across multiple servers. Sharding is particularly useful when a database table is too large to fit on a single instance, allowing the distribution of the workload, such as partitioning data by geographic regions. Replication enhances fault tolerance and contributes to scaling by maintaining multiple copies of the data across different servers.

Figure 2.8 illustrates an example architecture using ClickHouse's distributed table engine.

The architecture consists of three nodes:

- **Node 1 and Node 2**: These serve as both ClickHouse servers and ClickHouse Keeper servers.
- **Node 3**: This functions solely as a ClickHouse Keeper.

ClickHouse Keeper is an open-source coordination system, similar to the well-known system ZooKeeper. In a typical production environment, all Keeper services would be hosted on dedicated nodes to ensure optimal performance and reliability. However, even with a simple setup as illustrated in 2.8, it is possible to make use of the benefits of distributed query processing.

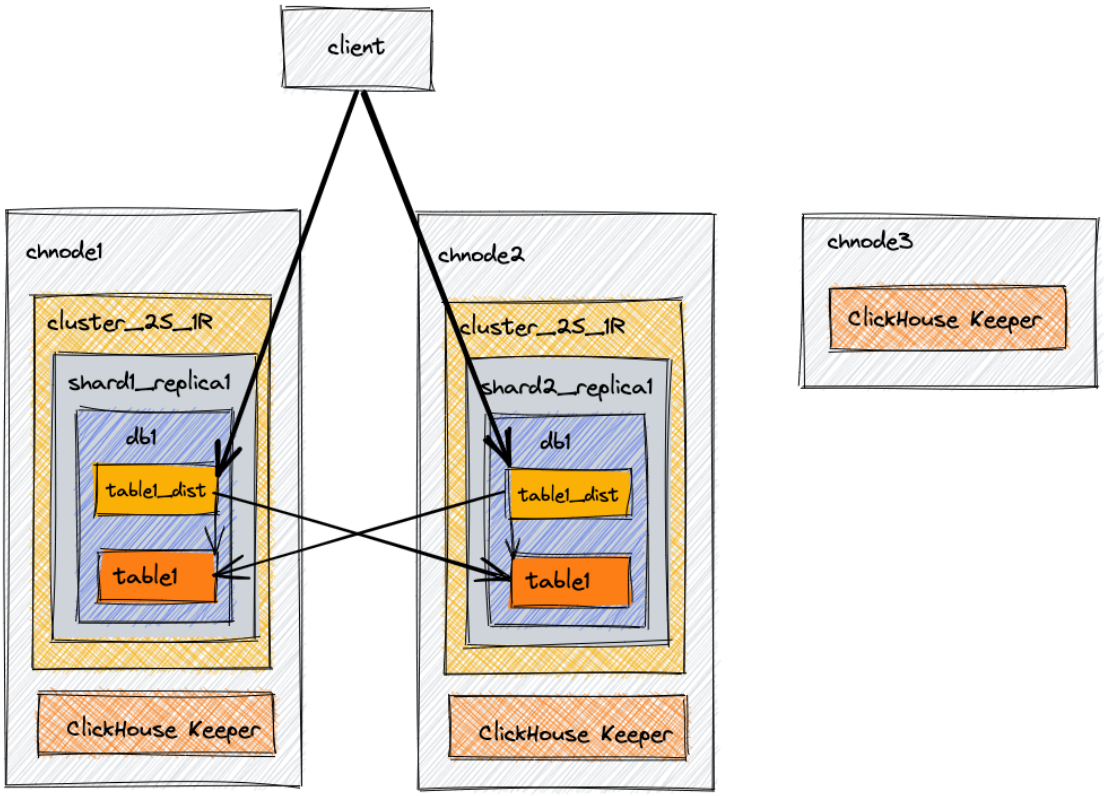


Figure 2.8: Distributed Processing in ClickHouse: Example Architecture [20]

2.4.3 Limitations

Although it was shown that ClickHouse provides a wide range of unique features, there are also certain limitations. One thing that can be considered a disadvantage is that it lacks full-fledged transactions. Also, it has limited capability to modify or delete already inserted data at a high rate and low latency (only batch updates and deletes are available for tasks such as GDPR compliance).

2.5 Column-Stores in Practice

In this section the different use cases for column-stores are explored, specifically, it will be discussed that they have shown to be advantageous in different areas of data management, including storage of semantic data.

2.5.1 Use Cases

The general consent is that data warehouses and analytical processing are the most common spaces where column-stores are used and where it is shown that they can outperform row-stores by more than an order of magnitude [22], [23]. This also aligns with where the focus of popular column-stores such as MonetDB, C-Store, or Vectorwise is laid, namely read-dominated workloads with less frequent updates that are mostly appending new data [33], [41], [46].

With the increasing popularity of fields that deal with exactly these types of read-dominated workloads, new use cases continue to emerge where column-stores show to be advantageous. Examples include the use of a column-store system, namely C-Store, for RDF (Resource Description Framework) data management [24], [40]. The papers show that the architecture of the C-store DBMS is very suitable for storing semantic data since data attributes from the same domain can benefit a lot from the compression techniques used in C-store. The results are very promising, with a reported performance improvement by a factor of 32 compared to the current state-of-the-art triple store [40].

Another interesting use case where column-stores excel can be found in [22]. The authors looked into using column-stores for **wide** and **sparse data**.

For **wide data**, e.g., tables with multiple thousands of attributes, the main observation that was made is that oftentimes, only a small number of those attributes is frequently accessed. For row-stores, this is a huge performance bottleneck since scanning a row with this enormous amount of attributes is very expensive. For column stores, on the other hand, the total number of attributes of a table does not impact the read performance of a fixed number of columns.

Sparse data benefits a lot from the compression features that many column-stores offer. Depending on the characteristics of the data, a different compression algorithm can be chosen for certain columns. In the specific case of **sparse data**, an appropriate NULL suppression algorithm, depending on the sparsity of the data, can be used. With these two observations in mind, the authors looked into applications where this type of data is prominent and found that column-stores may be a good choice for example in the case of Web data as well as XML data.

With this in mind, it is evident that column-stores can efficiently handle various types of data and workloads. While the analytics space remains the most common use case for

these systems, other common application scenarios are emerging, and the adoption of column-stores is likely to increase, driving further advancements and optimizations in this field.

CHAPTER 3

Join Query Optimization: Traditional to Structure-Guided

In this chapter, the concept of structure-guided query optimization is introduced. The aim is to provide the necessary theoretical background for the practical part of this thesis later in this thesis.

Section 3.1 begins by exploring traditional join processing. Specifically, it is explained how multiple relations are joined together in a DBMS, and what limitations these approaches have, particularly the issue of intermediate results explosion. Following this, Section 3.2 dives into the general idea of query optimization, explaining how these techniques enhance the efficiency of join queries and identifying the remaining challenges in join query optimization.

Finally, Section 3.3 focuses on structure-guided query optimization, and explains how these techniques can help us with the problem of large intermediate results.

The primary references for the theoretical background on query optimization and join processing include [27] and [39].

3.1 Traditional Join Processing

In this section, it will be explained how DBMS perform join operations in practice. For this, Section 3.1.1 talks about popular join algorithms that are often used in DBMS, Section 3.1.2 explains how these algorithms are used for multiple relation joins, and 3.1.3 talks about limitations and problems with these approaches.

3.1.1 Join Algorithms

This section elaborates on different implementations for performing a join operation in a DBMS. In 1993, a survey [32] was published, discussing three different algorithms, namely the nested loop join, the merge join, and the hash join, among other things. These three algorithms are fundamental and can still be found in most of today's DBMS that support a join operation. They will be explained in the following.

Nested Loop Joins

The **Nested Loop Join** is a straightforward join algorithm. To perform a natural join of two relations R and S, the algorithm iterates through all elements of the first relation R (the outer relation) and scans the second relation S (the inner relation) for matching elements. This process involves two nested loops: the outer loop for relation R and the inner loop for relation S, from which the algorithm derives its name.

Despite its simplicity, this join implementation suffers from poor runtime performance. Since the algorithm must find all matching pairs, for every element in the outer relation, the inner relation needs to be scanned from start to end, resulting in a runtime complexity of $O(n \times m)$, assuming n is the size of relation R and m is the size of relation S. This inefficiency makes the nested loop join impractical for large datasets. However, its memory consumption is minimal, as it only requires memory for the output data, making it suitable for certain scenarios. Various techniques can be used to enhance the efficiency of this approach.

One such technique is the **Block Nested Loop Join**. In this variant, instead of processing one tuple of the outer relation R at a time, an entire page of tuples from R is processed in each iteration. This reduces the number of scans required over the inner relation S, as multiple tuples from R are compared against S in a single pass. By using larger blocks that fit into the available memory, the algorithm can significantly reduce the number of disk I/O operations, thereby improving overall performance. The block nested loop join maintains the simplicity of the basic nested loop join while offering a more practical solution for larger datasets by optimizing the use of available memory resources.

Merge Joins

The **Merge Join** is often referred to as **Sort Merge Join** in some literature. This is because of the requirement of the algorithm to have its input data sorted on the join attributes, which can be either done by explicitly sorting the input data or sorting the data in advance. In 2.2.5, it was already mentioned that column-stores often store data sorted on different attributes, which can be a huge performance gain for precisely these types of operations.

The system can perform an interleaved linear scan with the sorted input data and find all matching elements accordingly. The performance of the merge join for a natural join

$R \bowtie S$ will be $O(n * log(n)$ for sorting relation R with input size n and $O(m * log(m))$ for sorting relation S with input size m, the actual join on the sorted input can then be done in $O(n + m)$ assuming that at least one of the relations is duplicate free.

With this in mind, it can be seen that merge join is especially suitable if the relations that need to be joined are already sorted since $O(n + m)$ is the fastest way possible to perform a join operation.

Hash Joins

The Hash Join algorithm's main idea is to build an in-memory hash table for the smaller input relation, referred to as the build input, using the join attributes as keys and the tuples as values. In the so-called probe phase, the algorithm performs a linear scan of the larger relation. While doing so it makes use of the previously built hash table to quickly find matching tuples by using the same hash function.

When the hash table fits in memory, the algorithm operates efficiently without needing temporary files or additional disk reads. The combined runtime complexity for the build and probe phases is $O(n + m)$, where n is the size of the build input and m is the size of the probe input.

However, in cases where the hash table cannot fit into memory, a partitioning approach is used. This ensures that matching tuples reside within corresponding partitions from both relations. Each partition of the smaller relation is read into memory, and the corresponding partition of the larger relation is scanned for matches. This method ensures that memory constraints are respected while still allowing for efficient joins, as partitions are only scanned once during the join process.

This partitioning approach is similar to the previously described idea behind the block nested loop join algorithm, where the join is processed in smaller chunks that fit into memory.

3.1.2 Multiple-Relation Join

All of the join implementations discussed in Section 3.1.1 perform a Two-Way-Join and are often referred to as binary join algorithms, meaning they deal with joining exactly two relations at a time.

There do exist database systems that implement other types of join algorithms, which can join more than two relations together at a time, e.g., DBMS that implement the worst-case optimal join [28]. However, most popular DBMS including PostgreSQL and ClickHouse, will perform several Two-Way joins if there are more than two relations in the FROM clause and utilize one of the join algorithms mentioned in Section 3.1.1.

3.1.3 Limitations of Traditional Join Processing

Inner joins have the property of being both commutative and associative. To illustrate the effect of this property on how DBMS perform multiple-relation joins and why this is a problem in the context of query optimization, let us consider the example of performing a join over a so-called triangle query:

$$R(A, B) \bowtie S(B, C) \bowtie T(A, C)$$

What a typical DBMS will do with such a query is that it will split the query into two pairwise joins using one out of the three algorithms mentioned in 3.1.1. For this specific example, the system has to choose between three different join orderings, which can be seen in Figure 3.1, and for the two join operations, it can choose one of the three possible join algorithms.

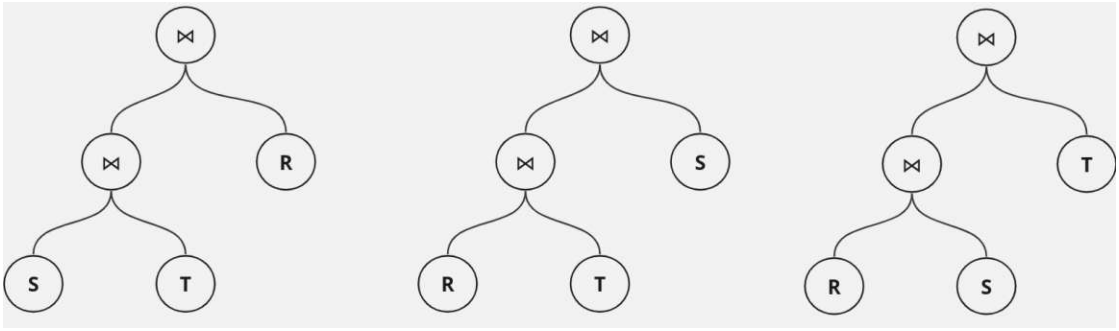


Figure 3.1: Join orderings for triangle Query $R \bowtie S \bowtie T$

Although the result of the three orderings will be the same in the end, the produced intermediate results might differ dramatically. This can be seen if the first two tables that are joined together consist of many tuples while the last table in the join sequence is significantly smaller. For example, R and S are large tables with millions of rows, and T is a much smaller table with a couple hundred rows. The join $R \bowtie S$ could result in an intermediate table that is unnecessarily large. This large intermediate table consumes significant memory and processing resources and increases the time for the subsequent join operation with relation S.

In the following section, it will be shown how query optimization can help to minimize execution time and computational resources by reducing unnecessarily large intermediate results. However, it will also be shown that although there exist numerous approaches aimed at mitigating the problem of intermediate result explosion, it still remains challenging to completely avoid this issue.

3.2 Query Optimization

Query optimization is all about finding the "best" query plan for a query [27]. Best in this context can refer to many things. However, typically, one wants to find a query plan

that has the minimum execution time with minimal costs so that users get their response as quickly as possible with minimal computational resources. A naive way to find the best query plan would be to enumerate all possible execution plans, determine the cost and execution time of each plan, and choose the best out of these.

Practically, this approach of considering all possible execution plans is insufficient for many reasons. One would be that the number of possible plans may be too big, and computing all these enumerations introduces a lot of overhead. Another problem is that even for a small number of plans, precisely determining the costs of a query plan would require fully executing it, which would again heavily slow down the processing of a query.

An important finding here is that query optimization techniques must be extremely fast. This is because all time spent on optimizing the query plan could also be spent on the actual query execution, and all the overhead that is introduced by query optimization will directly add up to the response time. Therefore, in practice, it is often sufficient to look only into several promising query plans, and instead of computing the costs, various estimation techniques are used.

In the following section, some basic query optimization techniques will be discussed. Specifically, it will also illustrate how optimizations related to the join operation are currently performed.

3.2.1 Transformative Optimization

In most RDMS, query plans are represented as a tree. Every node in the tree represents a query operation, and an edge between two nodes indicates the relationship between parent and child operators. Important here is that typically, there exist many different representations of a query as a query plan, all of which produce the same result and are semantically equivalent.

One common optimization technique is to take the initial plan that is generated by the DBMS and transform it into an equivalent, more efficient representation by various transformation rules. A key rule used during this process is to execute operations that reduce the size of intermediate results as early as possible. In practice, this means performing SELECT and PROJECT operations first to minimize the number of tuples and attributes involved in subsequent operations.

Additionally, the DBMS estimates the restrictiveness of SELECT and JOIN operations. Based on these estimates, it prioritizes operations that are expected to produce the fewest number of tuples. This approach ensures that the order of operations is optimized to minimize the size of intermediate results, thereby enhancing the overall efficiency of query execution.

3.2.2 Cost-Based Optimization

It was already mentioned in the introduction of this section that to compare different query plans e.g. the ones produced by a transformative optimizer, one still needs to somehow compare the costs of different query plans and choose the one with the lowest costs. The problem was that precisely computing the actual costs is very heavy in computation and, as a result, practically insufficient. Therefore, cost-based optimization techniques rely on cost estimates, and the more accurate these estimates are, the better a comparison of different plans can be performed.

To come up with a reasonably accurate cost function, many different costs are considered, including the following:

- access to secondary storage
- computation costs
- memory usage costs
- disk storage costs
- communication costs

A critical parameter used for estimating these types of costs is the so-called **selectivity**. Selectivity refers to the fraction of records that satisfy a specific condition and can be estimated using an attribute's number of distinct values. For this, one assumption must be made that the range of values is distributed uniformly. An essential tool used to improve the quality of estimates and produce reasonably accurate estimates in the case of non-uniformly distributed data is the use of **histograms**. These histograms divide the range of possible values for a particular attribute into so-called buckets. These buckets are then stored for important attributes and can help to get better estimations for the selectivity since it gives the DBMS an idea of how the data is distributed.

3.2.3 Join Ordering

Join ordering plays a crucial role in addressing the challenge of producing excessively large intermediate results during query execution. One issue here is that the commutative and associative properties of the join operation cause the number of potential query plans to escalate quickly as the number of join operators increases. This high number of plans makes it therefore computationally infeasible to compare all possibilities.

From a mathematical standpoint, this is analogous to the concept of permutations. For a query involving the joining of n relations, the number of possible orderings for these joins is given by the factorial $n!$. This factorial growth implies that the count of distinct query plans to evaluate becomes exceedingly large, even for relatively small values of n (e.g., $n = 6$ results in 720 different orderings).

As practically it is not feasible to consider all possibilities, query optimizers typically focus on a specific subset of query plans and evaluate only a limited selection from the vast range of possibilities. While this approach may sometimes miss the optimal plan, it dramatically simplifies the optimization process and often results in sufficiently efficient execution plans.

The following sections examine different tree structures to illustrate such constraints in join tree structures.

Join Tree Structures

In Figure 3.2 three popular join query tree structures are illustrated namely the left-deep join tree (3.2a), right-deep join tree (3.2b), and the bushy query tree (3.2c).

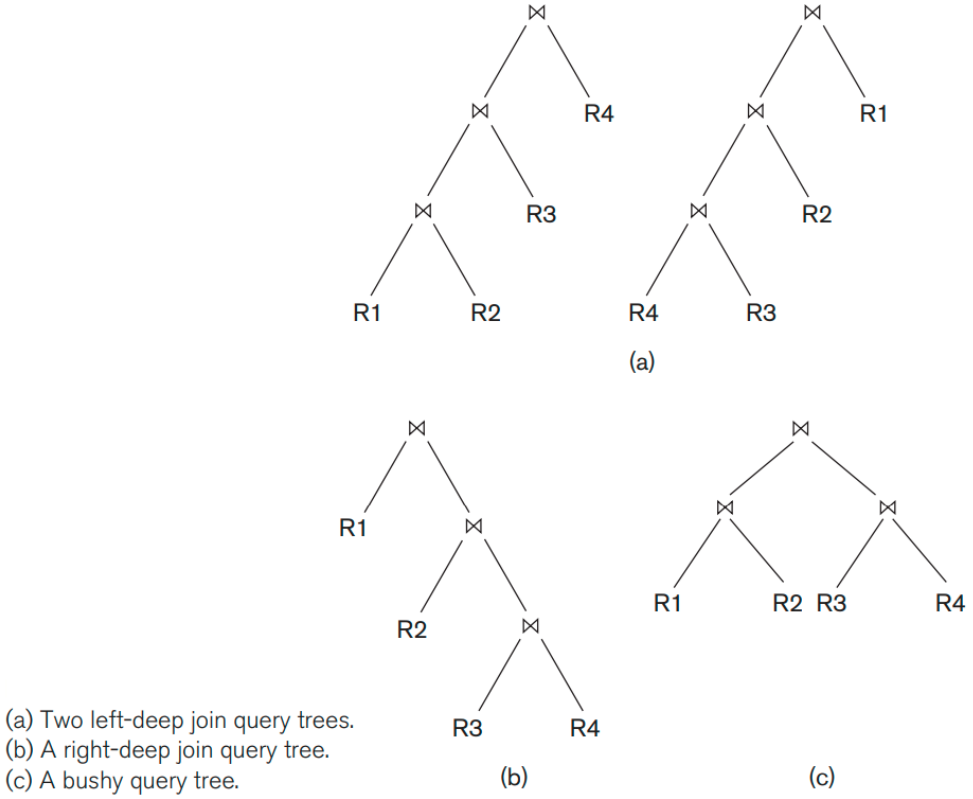


Figure 3.2: Join Tree Structures [27]

The one thing that all three types have in common is that they are considered binary trees, meaning that every parent has at most two children. A **left-deep tree** is a binary tree with the additional property that each non-leaf node's right child is a base relation. **Right-deep trees** are the mirrored version of left-deep trees and have the property that the left child of every leaf node is a base relation. The third common type of join

tree structures are **bushy query trees**. These are more complex binary trees where either the left or right child of an internal node can be another internal node. They offer a greater variety of shapes and link sequences, allowing for greater query optimization flexibility. However, the number of possible bush tree structures proliferates with the number of relationships, making it more challenging to find the optimal structure. This can also be seen in Figure 3.3, where the number of permutations is compared for the different join tree types.

No. of Relations N	No. of Left-Deep Trees $N!$	No. of Bushy Shapes $S(N)$	No. of Bushy Trees $(2N-2)!/(N-1)!$
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280

Figure 3.3: Number of Permutations for n Relations [27]

3.2.4 Limitations of Query Optimization Techniques

It has been demonstrated that techniques such as cost-based optimization help DBMS mitigate some of the limitations of join processing described in 3.1.3, allowing such queries to be solved in a reasonable time. It is important to note that while these techniques often find a well-optimized solution, they do not necessarily guarantee the most optimal solution.

Despite extensive research over many years, this fundamental compromise remains: One can either quickly compute a reasonably effective solution or strive for the optimal solution, which can be time-consuming or impractical to achieve within a reasonable timeframe, especially when dealing with many relations.

For this reason, in the next section, we will introduce a novel approach to address the described problem. We will demonstrate how this approach can be effectively leveraged to enhance query performance in practical scenarios.

3.3 Structure-Guided Query Optimization

In this chapter, the concept of structure-guided query optimization is introduced. Unlike the conventional optimization methods discussed earlier, this approach does not rely on cardinality estimates. Instead, it makes use of certain structural properties of a query.

This chapter begins by introducing key definitions related to the structural properties of queries. Specifically, it covers the definition of a Conjunctive Query (CQ) in Section 3.3.1, the concept of a Join Tree in Section 3.3.2, and talks about the cyclicity of queries in Section 3.3.3.

3.3.1 Conjunctive Query

A Conjunctive Query (CQ) is an essential type of query in database theory, and arguably the most fundamental type of query. These queries are typically represented using relational algebra.

In relational algebra, conjunctive queries (CQs) correspond to select-project-join queries and therefore only use the operations select σ, project π, and join $\bowtie$ [30]. In SQL, conjunctive queries correspond to `SELECT FROM WHERE` queries, where the `WHERE` conditions contain only equalities.

The combined complexity of solving a conjunctive query is NP-complete. However, some classes of CQs can be computed in polynomial time, which will be discussed in the following.

3.3.2 Join Tree

Before looking into the concept of acyclic queries, it is helpful to first look into the definition of a join tree which is taken from [38].

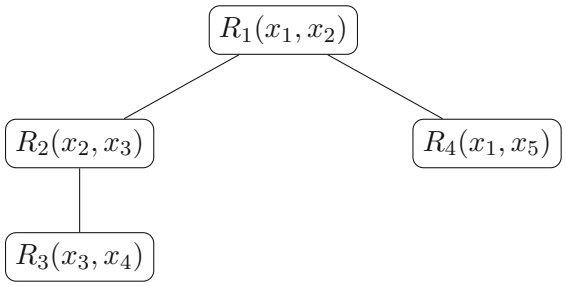
Let $Q(\vec{x}) \colon - R_1(\vec{z_1}), \ldots, R_n(\vec{z_n})$ be a Conjunctive Query (CQ). A join tree $T = (V, E)$ is a tree where:

- $V = \{R_1(\vec{z_1}), \ldots, R_n(\vec{z_n})\}$, i.e., V is the set of atoms in Q.
- E satisfies the condition that for all variables z of Q, the set $\{R_j(\vec{z_j}) \in V \mid z \text{ occurs in } R_j(\vec{z_j})\}$ induces a connected subtree in T.

To better understand how this definition translates to a query, consider the following example in Datalog notation:

$$Q_e(x_1, x_2, x_3, x_4, x_5) : -R_1(x_1, x_2), R_2(x_2, x_3), R_3(x_3, x_4), R_4(x_1, x_5)$$

A corresponding join tree can be represented as follows:



We can see from this example, that each node contains the query atoms, and the edges denote the join conditions between these relations based on the shared variables. This is called the running intersection property, which guarantees that for every query variable, the tree nodes that contain that variable must form a connected subgraph.

3.3.3 Acyclic CQs

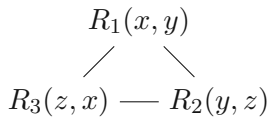
The primary objective of structure-guided query optimization is to find queries that have certain properties that allow us to solve those efficiently. Acyclic Conjunctive Queries (ACQs) satisfy exactly this.

These types of queries are significant in database theory due to those mentioned favorable computational properties, meaning they can be evaluated in polynomial time. Formally, a conjunctive query Q is considered acyclic if it has a join tree [38].

A very often-used example of a cyclic conjunctive query is the triangle query. It forms a cycle involving three relations, each joining the other two. The query Q_{tr}:

$$Q_{tr}(x, y, z) : -R_1(x, y), R_2(y, z), R_3(z, x)$$

can be represented as:



This diagram illustrates the cyclic nature of the triangle query, where each relation connects to the other two, forming a closed loop. Using the definition from 3.3.2, it is impossible to form a valid join tree for this query.

3.3.4 Finding and Evaluating ACQs

We have already identified in Section 3.3.3 that Acyclic Conjunctive Queries (ACQs) can be evaluated efficiently. However, two critical questions remain:

1. Since ACQs are defined by the existence of a join tree, how can we efficiently find a join tree for such a query and thus decide if it is indeed acyclic?
2. Once an ACQ is identified, how can the query be evaluated efficiently?

Finding a join tree

A solution to the first problem stated is the so-called **GYO-reduction**, which is named after the authors Graham, Yu, and Ozsoyoglu, who independently came up with a similar

idea in [45] and [36]. This algorithm can efficiently decide if a CQ is acyclic and if so also produce one possible join tree. There exist different variants of the GYO-reduction, the one used in the practical part of this thesis was introduced in [30] and follows the pseudocde in Figure 3.4.

input : A connected α-acyclic hypergraph H
output: A join tree of H
1 $J \leftarrow$ empty tree;
2 **while** H *contains more than 1 edge* **do**
3 Delete all degree 1 vertices from H;
4 **for** $e \in E(H)$ *s.t. there is no* $f \in E(H)$ *with* $e \subset f$ **do**
5 $C_e \leftarrow \{c \in E(H) \mid c \subseteq e\}$;
6 **for** $c \in C_e$ **do**
7 Set label(c) as child of label(e) in J;
8 Remove c from H;
9 **return** J;

Figure 3.4: Flat-GYO algorithm [30]

Notice, that the input for this algorithm is a so-called hypergraph. A hypergraph is a generalization of a graph in which an edge can join any number of vertices. In database theory, a hypergraph is often used to represent the structure of a conjunctive query. The vertices represent attributes (also referred to as variables), and hyperedges represent relations that include those attributes.

To see how such a representation of a CQ using a hypergraph works, let us consider the following example query:

$$Q_{hg}(x, y, z, p) : -R_1(x, y, z), R_2(y, p), R_3(y, z, p)$$

The corresponding hypergraph representation of query Q_e is given in Figure 3.5. The representation uses rectangles to represent the relations R_1 (blue), R_2 (red), and R_3 (yellow). The variables x, y, z, p are represented as circles.

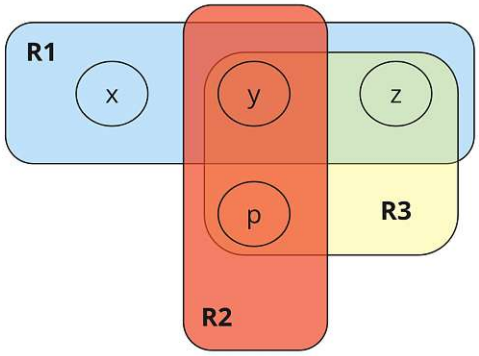


Figure 3.5: Q_{hg} represented as Hypergraph

Yannakakis' Algorithm

Up to this point, we have discussed that Acyclic Queries (AQs) can be solved efficiently. In Section 3.3.4, we demonstrated that by using GYO-reduction, we can both determine whether a given query is acyclic and, if so, construct a join tree.

The remaining question is: how can ACQs be solved efficiently?

The answer to this is the so-called Yannakakis' algorithm [44]. Despite its age, the algorithm addresses a crucial problem: preventing the explosion of intermediate results, thus making it feasible to solve large joins that would otherwise be challenging or impossible to handle within a reasonable time frame.

Yannakakis' algorithm operates in three join phases. The join conditions are always determined by an intersection of the respective nodes attributes/columns in the join.

Given a join tree of an ACQ, the algorithm proceeds through the following steps:

1. **Bottom-up Semi-Join Traversal: Upwards Propagation**
   In the initial step, the algorithm performs a bottom-up traversal of the join tree to eliminate dangling tuples—those that do not contribute to the final result. This is achieved through semi-join operations, where each parent node performs a semi-join with its child nodes ($parent \ltimes child$), starting from the leaves.

2. **Top-Down Semi-Join Traversal: Downwards Propagation**
   The second step involves a top-down traversal and performing the semi-joins in reversed order. During this phase, the algorithm refines the results obtained from the bottom-up traversal by propagating solutions downwards through the join tree, meaning that only those contributing to the query result are retained. This process involves each child node performing a semi-join with its parent node: $child \ltimes parent$

3. **Second Bottom-Up Traversal: Compute Solutions Using Joins**
   The final step uses a second bottom-up traversal where the algorithm computes

the final solutions using join operations. Join operations combine the intermediate results obtained from the semi-join reduction steps to derive the overall result of the conjunctive query. This step completes the enumeration process and provides the desired output for the given acyclic query.

For any given database instance D, by applying these steps on ACQs, it is possible to compute an answer to the query in output polynomial time.

Another important finding is that during the first step of Yannakakis' algorithm, namely the bottom-up traversal if any node in the join tree produces an empty intermediate result, the final result will also be empty.

This applies to all subsequent stages of Yannakakis' algorithm. Thus, if an intermediate result for any node during the traversal is empty, the final result of the entire query will be empty as well.

This means that if we only are interested in whether the query will be empty or not, which is naturally the case for boolean queries, the computation is feasible in polynomial time, and can be done by only applying the first step of Yannakakis' algorithm.

Yannakakis' Algorithm Example

To better understand the three steps of Yannakakis' algorithm, let us take a look at a practical example. Consider the following relations with their corresponding attributes:

- $R_1(x_1, x_2, x_3)$
- $R_2(x_2, x_3)$
- $R_3(x_3)$
- $R_4(x_2, x_4, x_3)$

Each relation contains tuples. The indices in the tuples refer to specific instances or rows in the relation. For example, $t_{1,1}$ denotes the first tuple in relation to R_1. Here are the tuples for each relation:

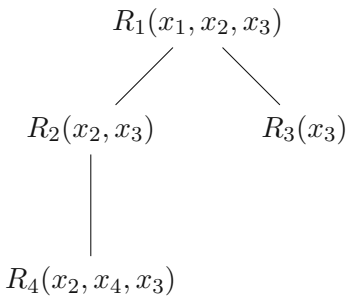
Tuple	x_1	x_2	x_3
$t_{1,1}$	s_1	c_1	b_1
$t_{1,2}$	s_1	c_1	b_2
$t_{1,3}$	s_3	c_3	b_1
$t_{1,4}$	s_3	c_1	b_4
$t_{1,5}$	s_2	c_2	b_3

Tuple	x_2	x_3
$t_{2,1}$	c_1	b_2
$t_{2,2}$	c_1	b_1
$t_{2,3}$	c_4	b_6

Tuple	x_3
$t_{3,1}$	b_1
$t_{3,2}$	b_2

Tuple	x_2	x_4	x_3
$t_{4,1}$	c_1	a_1	b_1
$t_{4,2}$	c_1	a_1	b_2
$t_{4,3}$	c_1	a_2	b_2

The given join tree for these relations that serves as input for the Yannakakis' algorithm is:



Since we have now clarified the input for Yannakakis' algorithm, we can apply the three steps as follows:

1. **Bottom-up semi-joins:**
   - $R_2' = R_2 \ltimes R_4$

Tuple	x_2	x_3
$t_{2,1}$	c_1	b_2
$t_{2,2}$	c_1	b_1

     Removed tuple: $t_{2,3} = (c_4, b_6)$ since $x_2 = c_4$ and $x_3 = b_6$ do not match any x_2, x_3 pairs in R_4.
   - $R_1' = (R_1 \ltimes R_2') \ltimes R_3$

Tuple	x_1	x_2	x_3
$t_{1,1}$	s_1	c_1	b_1
$t_{1,2}$	s_1	c_1	b_2

     Removed tuples: $t_{1,3}, t_{1,4}$ and $t_{1,5}$.

2. **Top-down semi-joins:**

   - $R_2'' = R_2' \ltimes R_1'$:

Tuple	x_2	x_3
$t_{2,1}$	c_1	b_2
$t_{2,2}$	c_1	b_1

     No tuples removed as remaining $t_{2,1}, t_{2,2}$ pairs in R_2' match those in R_1'.

   - $R_4' = R_4 \ltimes R_2''$:

Tuple	x_2	x_4	x_3
$t_{4,1}$	c_1	a_1	b_1
$t_{4,2}$	c_1	a_1	b_2
$t_{4,3}$	c_1	a_2	b_2

     No tuples removed as remaining $t_{4,1}, t_{4,2}, \text{and} t_{4,3}$ pairs in R_4' match those in R_2''.

   - $R_3' = R_3 \ltimes R_1'$:

Tuple	x_3
$t_{3,1}$	b_2
$t_{3,2}$	b_1

     No tuples removed as remaining $t_{3,1}$ and $t_{3,2}$ pairs match those in R_1'.

Step 3: Final Join

Perform the final join on the reduced relations:

$$\pi_{x_1,x_2,x_3,x_4}(R_1' \bowtie R_2'' \bowtie R_3' \bowtie R_4')$$

x_1	x_2	x_3	x_4
s_1	c_1	b_1	a_1
s_1	c_1	b_2	a_1
s_1	c_1	b_2	a_2

0MA Queries

In [30] a special class of queries known as zero-materialization answerable (0MA) queries was first introduced. These queries are unique because they can be resolved by performing only the initial bottom-up semi-join phase of Yannakakis' Algorithm.

To characterize these 0MA queries, we first need to define a few concepts. In relational algebra, aggregation operations, typically represented by the `GROUP BY` clause in SQL, are denoted as $\gamma_U(\cdot)$, where U specifies the columns involved in the aggregation. A query in aggregation normal form can be expressed as $Q = \gamma_U(\pi_S(Q'))$, where Q' involves only natural joins and selection operations.

For a query to be classified as 0MA, it must satisfy the following conditions:

1. **Aggregation Normal Form:** The query should be in the form $Q = \gamma_U(\pi_S(Q'))$, where Q' contains only joins and selections.
2. **Guarded Query:** There must exist a relation R that includes all attributes specified in the output and the `GROUP BY` clause. This relation R is said to "guard" the query.
3. **Set-Safe:** The query must be set-safe, meaning the result should remain consistent whether or not duplicates are removed before aggregation.

These criteria ensure that the query can be evaluated without generating intermediate join results. This is particularly advantageous for aggregate functions like `MIN` and `MAX`, which are inherently set-safe. Additionally, `COUNT` can achieve set-safety when combined with `DISTINCT`.

The main advantage of 0MA queries is the significant reduction in computational overhead, especially for large datasets with complex relational structures. This efficiency is achieved by leveraging the structure of the query to minimize unnecessary data processing.

Application of Structure-Guided Query Optimization

Traditional optimization strategies discussed in Section 3.2 have been studied for a long time. These optimizes, while generally efficient, often fall short in providing a guaranteed bound on query-answering time.

The key insight from Yannakakis' algorithm is that for queries with acyclic hypergraphs, it is possible that answers to such queries can be computed in polynomial time with respect to the input and output size. This is very different from traditional optimizers since the Yannakakis' algorithm guarantees an upper bound on query answering and secondly does not rely on any quantitative metrics, but instead exploits the structural properties of a query.

Although, from a theoretical standpoint this polynomial upper bound is extremely promising, structure-guided query optimization has not found its way into mainstream database systems. In the main part of this thesis 4, the aim is to investigate why exactly this is the case, and if the specific investigated database system (Clickhouse) can benefit from such an integration.

CHAPTER 4

Integrating a Structure-Guided Query Optimizer into ClickHouse

In the following the integration of a structure-guided query optimizer into the column store database system ClickHouse is presented. This optimizer implements the first bottom-up traversal of Yannakakis' algorithm and is specifically designed for 0MA queries as defined in Section 3.3.4. The source code for the optimizer implementation can be found on Github [14].

We start by looking into how ClickHouse processes queries and how they are internally represented in Section 4.1. Next, we address the challenges and considerations encountered during the integration process, why certain decisions were taken, and provide insights not covered in the official documentation. Finally, in Section 4.3 the actual integration is shown in detail.

4.1 ClickHouse Query Processing

This section aims to give an overview of how queries in ClickHouse are executed. This includes how queries are internally represented, how they can be optimized, and what important interfaces and classes are used during the execution. Most of these insights come from reverse engineering the ClickHouse codebase [6] and looking into the corresponding documentation.

In Section 4.1.1 an overview of relevant steps that happen and classes used during query processing is given. Section 4.1.2 goes more into detail about how queries in ClickHouse are parsed and internally represented. It is important to note that this overview of how ClickHouse processes queries is greatly simplified and the focus lies in understanding the parts that are relevant for the practical part of this thesis.

To illustrate the query execution process, the following example query Q_{ex} will be used throughout this section:

```
SELECT *
FROM
    insert_select_testtable1,
    insert_select_testtable2,
    insert_select_testtable3
WHERE
    insert_select_testtable1.a = insert_select_testtable2.a
    AND insert_select_testtable1.a = insert_select_testtable3.a
```

Listing 4.1: Example query Q_{ex}

The tables were instantiated with the following query, using the MergeTree engine:

```
CREATE TABLE insert_select_testtable*
(
    `a` Int8,
    `b` String,
    `c` Int8
)
ENGINE = MergeTree()
```

Listing 4.2: Create statement used for tables in Q_{ex}

4.1.1 Query Processing Overview

One of the most central classes for query execution in ClickHouse is `ExecuteQuery.cpp`. From there most of the other underlying steps are taken. An overview of some of the relevant classes that are used during this process can be found in Figure 4.1.

When the query is sent from one of the possible ClickHouse clients, the ClickHouse server engine will receive the query as a string. `ExecuteQuery.cpp` will then call a corresponding parser for the query, that will go over the query string and turn it into an AST structure. More on how this is done will be explained in Section 4.1.2.

After parsing the query and constructing the AST structure, a corresponding interpreter will be called. For our example query this would be `InterpreterSelectQuery.cpp`. This class takes as input the AST structure, does certain query optimizations, and returns a query pipeline. InterpreterSelectQuery also uses a class called ExpressionAnalyzer. This class does most of the rule-based optimizations. The query pipeline is then executed as part of the interpreter class and will produce in the end the output that is returned to the user. More details on what happens in the step of query interpretation and how query pipelines in ClickHouse work will be given in Section 4.1.4.

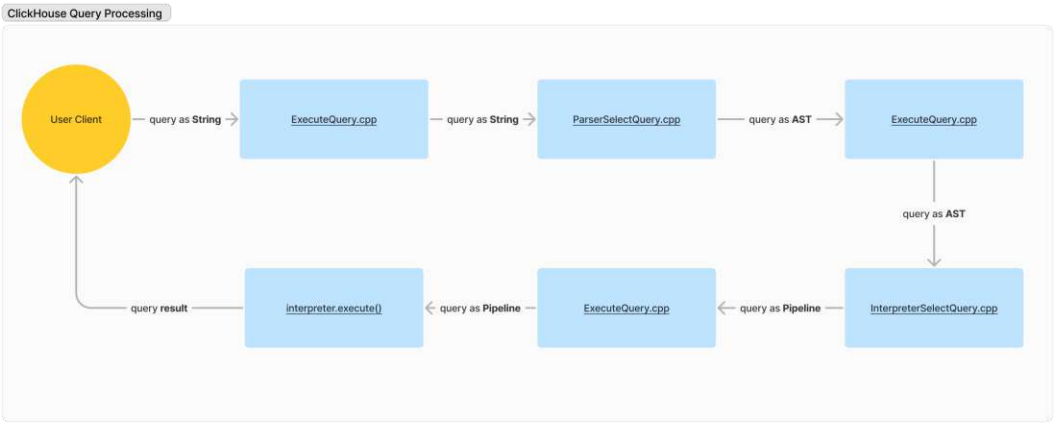


Figure 4.1: Query Processing High-Level Overview

As of today, according to the developers they are working on a new interpreter called `InterpreterSelectQueryAnalyzer.cpp`. The idea is that with this interpreter a new abstraction between AST and QueryPipeline is introduced, the so-called QueryTree. Currently, this is not yet used in production but can be manually activated using a specific flag.

4.1.2 Query Parsing

The first step in processing the query is parsing the user query, which is given as a string, and translating it into an internal representation.

There are different kinds of parsers, and most of them work in a recursive manner. The most relevant one, and also the one that will be used for parsing our example query is `ParserSelectQuery.cpp`.

The class will go over the input query and look for certain keywords e.g. `SELECT` and call another underlying parser that will deal with the corresponding part of the query.

The different parsers systematically traverse each part of the query and convert each segment into an abstract syntax tree (AST), which serves as the internal representation of the query structure.

4.1.3 AST

An Abstract Syntax Tree (AST) is a tree data structure and is very often used in the context of compilers.

In ClickHouse, the AST serves as the main data structure for internally representing and interpreting a query. Queries and all subparts of a query are represented as AST nodes, which are instances of IAST. Each AST node always has an attribute `children`, which

is a list of other AST nodes. To examine this representation in more detail, let's look at an AST representation of our example query Q_{ex}:

```
SelectQuery, 0x00007f058c04e918
-ExpressionList, 0x00007f06b346b6f8
--Asterisk, 0x00007f057cc45a18
-TablesInSelectQuery, 0x00007f057cc9b218
--TablesInSelectElement, 0x00007f058c04e818
---TableExpression, 0x00007f0576671b58
----TableIdentifier_insert_select_testtable1, 0x00007f057cf
   14058
--TablesInSelectElement, 0x00007f058c1da618
---TableExpression, 0x00007f057cf133d8
----TableIdentifier_insert_select_testtable2, 0x00007f058c
   044198
---TableJoin, 0x00007f058c1da418
--TablesInSelectElement, 0x00007f05dfdbb418
---TableExpression, 0x00007f057cf13658
----TableIdentifier_insert_select_testtable3, 0x00007f057cf
   13158
---TableJoin, 0x00007f057cf5b618
-Function_and, 0x00007f057cf1a898
--ExpressionList, 0x00007f057cc45af8
---Function_equals, 0x00007f057cf1ad18
----ExpressionList, 0x00007f06b346b618
-----Identifier_insert_select_testtable1.a, 0x00007f058c0442d8
-----Identifier_insert_select_testtable2.a, 0x00007f057ce0d0d8
---Function_equals, 0x00007f059d7dfc18
----ExpressionList, 0x00007f06b346b7d8
-----Identifier_insert_select_testtable1.a, 0x00007f057cf13798
-----Identifier_insert_select_testtable3.a, 0x00007f057cf138d8
```

Listing 4.3: Example of AST structure

The representation in Listing 4.3 shows the hierarchical structure, where each line represents a node in the AST and its associated metadata: the type of AST element e.g. `SelectQuery` and its memory address. The tree structure is shown with indentation, where children are listed under their respective parent nodes, illustrating the hierarchical relationship among the elements of the AST.

Since understanding this representation is an essential part of how ClickHouse interprets and processes queries, in the following, some parts of the AST structure will be explained in detail.

ExpressionList

The `ExpressionList` element is a fundamental component within a `SelectQuery`, serving various purposes, including the selection of columns in a query. In our example query from Listing 4.2, the columns are selected using the wildcard asterisk symbol (*).

After the parsing is done, the AST will be handed over to an interpreter, where the `Asterisk` in the `ExpressionList` will be transformed. After the `ExpressionList` is transformed it contains a list of identifiers for corresponding columns.

Notice, that `ExpressionList` is quite a universal AST structure and is used not only for internally representing the `SELECT` part of our example query but also for the columns that are introduced as part of the `WHERE` clause.

TablesInSelectQuery and TablesInSelectElement

`TablesInSelectQuery` represents the `FROM` part of a query and contains at least one `TablesInSelectElement`. The `TablesInSelectElement` represents a resulting table or subquery with some additional information and has the following list of possible children:

- `TableExpression`: Represents the table itself or a subquery.
- `TableJoin`: Indicates how tables are joined if `TableExpression` is present.
- `ArrayJoin`: Specifies arrays to be joined. ClickHouse supports this operation for generating a new table where each row corresponds to an array element from the original column. It can be thought of as performing a JOIN operation with arrays or nested data structures, allowing for easy manipulation of array data within queries.

For the first element of the list of `TablesInSelectElement`, either `TableExpression` or `ArrayJoin` could be non-null. For subsequent elements, both `TableJoin` and `TableExpression` are non-null, or `ArrayJoin` is non-null.

This rule is quite important and it essentially guarantees, that except for the first table, all subsequent tables must have a corresponding join operation.

TableJoin

`TableJoin` is an element that specifies how tables are joined together in a query. It has the following list of possible attributes that further specify how the join operation is performed:

- `kind`: The type of join (e.g., INNER, LEFT, RIGHT).
- `strictness`: The join behavior when matching rows (e.g., ALL, ANY, SEMI).

- `clauses`: Conditions specifying how tables are joined, including key column names and comparison logic.
- `columns_from_joined_table`: List of columns that can be accessed from the joined table.

4.1.4 Query Interpreter

Interpreters in ClickHouse play an essential role during query processing. The main goal of it can be summarized as turning the query from an AST representation to a query pipeline.

There are various interpreters for different query types, with some of them being quite simple e.g. InterpreterDropQuery, and the more complex ones e.g. InterpreterSelectQuery.

The query execution pipeline uses so-called processors that are capable of handling and generating data in the form of chunks, which consist of columns with specific types. Processors communicate via ports, each having multiple input and output ports.

Depending on the type of query and interpreter used for this query, different types of pipelines will be constructed. For instance, interpreting a `SELECT` query yields a "pulling" QueryPipeline equipped with a specialized output port for retrieving the result set. On the other hand, interpreting an `INSERT` query results in a "pushing" QueryPipeline featuring an input port for inserting data. Meanwhile, interpreting an `INSERT SELECT` query produces a "completed" QueryPipeline.

4.1.5 Existing Optimizer

ClickHouse applies optimizations in the form of so-called “passes”. Each of these passes applies certain optimizations to the query and it is possible to investigate what happens during one of these passes using the `EXPLAIN` statement and specifying the `passes` parameter.

Most of the optimizations are done on the Query Plan. Examples of these types of optimizations include various predicate push-down optimizers such as tryPushDownLimit.

Some other optimizations are also done directly on the AST structure as part of the interpreter. An example of this would be the cross to inner join rewriter that is executed within the `InterpreterSelectQuery` interpreter.

4.2 Challenges and Considerations

In the following section, some of the challenges that were dealt with during the practical part of this thesis will be discussed. The aim is that this should give a better understanding of why certain steps were taken and what considerations were taken into account.

4.2.1 Reverse Engineering ClickHouse's Codebase

The first challenge that was dealt with before starting with the actual implementation was getting a better understanding of the codebase of ClickHouse. The main focus was to understand how ClickHouse translates queries into internal data structures and how the queries are then processed.

To achieve this, an extensive reverse engineering effort was undertaken. The initial steps involved executing a variety of test queries against a local ClickHouse deployment. Subsequently, looking into the generated logs provided insights into potentially relevant classes for the integration.

Although there also exists comprehensive documentation for the codebase, it lacks in giving insights into the bigger picture, especially on the internal representation of queries within ClickHouse. This exploration was therefore very important for understanding ClickHouse's query processing mechanism.

4.2.2 Rewrite of Joins

One important discovery that was made during reverse engineering of the code base is that for queries with more than one join operation, ClickHouse will rewrite the query using subqueries.

This happens using `JoinToSubqueryTransformMatcher.cpp` and is called within a method called `rewriteMultipleJoins` in `InterpreterSelectQueryAnalyzer.cpp`.

This transformation is done on the AST structure of the query and allows the processing of a query as a number of two-way joins of multiple subqueries. If we consider an example query like the one in Listing 4.4.

Listing 4.4: Original Query

```
1 SELECT ...
2 FROM tableA
3 JOIN tableB ON ...
4 JOIN tableC ON ...
```

The manipulated AST represented as a query can be seen in Listing 4.5.

Listing 4.5: Query after JoinToSubqueryTransformMatcher was applied

```
1 SELECT ...
2 FROM (
3     SELECT --.s.*, ...
4     FROM tableA
5     JOIN tableB ON ...
6 ) AS --.s
7 JOIN tableC ON ...
```

In this transformed query:

- The first join between tableA and tableB is wrapped in a subquery aliased as `-.s`.
- The necessary columns in the subquery are selected using an alias `-.s.*` to avoid alias clashes.
- The outer query joins the result of the subquery (–.s) with tableC using the original join condition.

Although there was no documentation found for this rewriter, this approach likely aligns with the conventional strategy of translating multiple join statements into a sequence of two-way joins, which would be an explanation for ClickHouse's way of using subqueries in these scenarios.

4.2.3 Experimental Analyzer

It was previously mentioned that the developers of ClickHouse are currently working on a new analyzer called `InterpreterSelectQueryAnalyzer.cpp`. One huge change that comes with this analyzer is the introduction of a new abstraction between `AST` and `QueryPlan`, the so-called `QueryTree`.

This analyzer switched as part of the ClickHouse release 24.3 LTS (2024-03-27) from experimental to beta status and can be turned on or off using a flag called `allow_experimental_analyzer`.

As of today this experimental analyzer is still under development and has potentially great impact on the way queries are processed in ClickHouse. This was an important consideration that was taken into account for the practical part of this thesis and will be explained in more detail in the next section.

4.2.4 Finding the right place for the modification

In Section 4.1.5 it was mentioned that most of the query optimization happens on the level of the query plan, with some exceptions that are also done directly on the AST structure.

In the initial implementation phase, the idea was to follow this convention and do all the modifications that need to be done as part of one optimizer on the level of the query plan that would be executed at the end. This also brings the benefit that the query is processed exactly the way we want it to be.

However, there were a few challenges that had to be faced and in the end, led to a different approach. The biggest problem was that the introduction of the new experimental analyzer described in Section 4.2.3, which is still under development, has a significant impact on the query representation as query plans. Also, it was uncertain if the newly

introduced abstraction level of a query tree would have any impact on our integration. Another issue was ClickHouse's behavior, as discussed in Section 4.2.2, where queries with multiple joins are translated into subqueries, altering table and column naming conventions.

After facing multiple issues, those coming from the experimental analyzer and the transformation of joins into subqueries, the decision was made to do all modifications to the codebase before the query execution process, in particular before the rewrite to subqueries. This meant in practice that rather than operating on the query plan directly, the optimizer would now manipulate the AST structure. While ClickHouse does feature optimizers, such as one that converts cross joins to inner joins, this adjustment introduced a drawback: subsequent optimizers in the query execution pipeline could potentially conflict with our modifications, resulting in a mismatch between the intended query plan and the final execution. As a consequence, it was important to validate for correctness of the optimizer and, if necessary, deactivate conflicting optimizers to avoid such discrepancies.

4.3 The Integration

In this section, the actual implementation and architecture of the integration will be outlined.

Generally, all manipulations that happen as part of this optimizer happen within one class `YannakakisOptimizer.cpp`. This class is called from the `ExecuteQuery.cpp` and manipulates the query in the form of an AST. As described in Section 4.2.4, the call to `YannakakisOptimizer.cpp` is done, before the rewrite to subqueries with `JoinToSubqueryTransformMatcher.cpp` happens.

The most important steps that are performed as part of this optimizer can be summarized as follows:

- Collect tables and predicates
- GYO reduction
- Reroot
- Bottom-up semi-join

In the following sections, the different steps will be explained in detail including what they do and why they are necessary.

4.3.1 Collect tables and predicates

The main goal of this step is to collect all necessary information from the input query and store those in suitable data structures. This is done within a function called

`collectTablesAndPredicates`. In this context, predicates could be elements such as join conditions or filtering attributes used as part of a `WHERE` clause.

Essentially, there are five important attributes that are used for storing all the necessary information from the query, which subsequently play important roles in the later stages of the optimization process.

- `tableWithPredicateNames`: A mapping that uses the table name as a string key, and holds a set of predicates of all predicates that are part of the ON clause or WHERE clause of that specific table.
- `tableObjects`: A mapping that has again the table name as a string key. This mapping connects to the corresponding table object within the original Abstract Syntax Tree (AST) through an ASTPtr.
- `predicateObjects`: Maps all join predicates with their identifier to an ASTPtr object that points to the corresponding predicate of the original AST.
- `selectionObjects`: Holding all filters (or selection in relational algebra) from the WHERE clause that are not part of a join condition. This is done with a mapping using the table name as a string key, and a vector containing all selection objects as its value.
- `disjointSet`: Holds all join predicates, more on this will be explained in Section 4.3.2.

Generally, the main goal of getting all the information from the query is achieved by traversing the AST of the query and updating the corresponding attributes. Most of this can be seen as trivial, except for two things that need to be considered.

First, how can one distinguish between predicates that are used as part of a join, and predicates that are used as part of a selection?

Second, how does one represent the join predicates such that in the later steps it is guaranteed that all of these predicates are the same? An example of this would be if we have two join operations such as `tableA.A == tableB.B AND tableB.B == tableC.C`, then we need to somehow efficiently store the information that all three attributes `A`, `B`, and `C` must be the same.

The first consideration that handles the differentiation between predicates that are part of a join and filtering predicates, is solved by looking into the origin of the arguments of the predicate. For example in the case of an equality check e.g. `tableA.A == tableB.B` one can see that `A` and `B` both originate from a table and therefore it can be concluded that the predicate is part of a join. In all other cases, the predicate must be part of a selection.

Regarding the second consideration, the idea was to use so-called equivalence classes.

4.3.2 Representing Join Predicates as Equivalence Classes

In order to represent equivalence classes for join predicates, a data structure known as a disjoint set (also known as union-find) was implemented and can be found in `DisjointSet.cpp`.

A disjoint set data structure maintains a collection of disjoint (non-overlapping) sets. It provides two main operations: `Union` and `Find`. The `Union` operation merges two sets together, while the `Find` operation determines which set a particular element belongs to.

In the context of representing join predicates as equivalence classes, each attribute involved in a join predicate is initially assigned to its own set. This initialization is done by calling a function `addToSet`, which can be seen in Listing 4.6.

Listing 4.6: Disjoint Set: Add Key

```
bool DisjointSet::addToSet(std::string key)
{
    if (parent.contains(key))
        return false;

    parent[key] = key;
    rank[key] = 1;

    return true;
}
```

The keys used in this function would in our case be the join conditions, and in the beginning, every join condition is pointing to its equivalence class.

When encountering a join predicate such as `tableA.A == tableB.B`, the `Union` operation is used to merge the sets containing `A` and `B`, indicating that they are equivalent. This can be seen in Listing 4.7.

Listing 4.7: Disjoint Set: Union

```
// Union two sets containing elements x and y
void DisjointSet::unionSets(std::string x, std::string y)
{
    if (!parent.contains(x))
        addToSet(x);
    if(!parent.contains(y))
        addToSet(y);
    std::string rootX = find(x);
    std::string rootY = find(y);

    if (rootX != rootY)
    {
```

```
13          // Union by rank to keep the tree balanced
14          if (rank[rootX] < rank[rootY])
15          {
16              parent[rootX] = rootY;
17          }
18          else if (rank[rootX] > rank[rootY])
19          {
20              parent[rootY] = rootX;
21          }
22          else
23          {
24              parent[rootY] = rootX;
25              rank[rootX]++;
26          }
27      }
28  }
```

For example, given the join predicates `tableA.A == tableB.B AND tableB.B == tableC.C`, after processing these predicates, all three attributes `A`, `B`, and `C` would belong to the same equivalence class.

This allows subsequent processing steps to efficiently identify that these attributes are related and must have the same value.

Notice, that the implementation also stores a "rank" for every node. This rank represents an upper bound of the height of a node and this value is used in the union function. The idea is that nodes with a higher rank will always be used as parents. This leads to a more balanced tree structure, making the find operation more efficient.

4.3.3 GYO reduction

The main goal of the GYO reduction is to produce a join tree, that will tell us later in which order the different join operations need to be executed.

As part of the integration, a variant of GYO was implemented called Flat-GYO. Initially introduced in [30], the implementation largely follows the already presented pseudocode that can be seen in Figure 3.4. This section focuses more on the supplementary steps undertaken during implementation.

All these steps are done within a single function called *gyoReduction*. This function takes as input parameters the attributes tablesAndPredicates (tables and corresponding predicates) and disjointSet (join predicates equivalence classes) computed in the previous step 4.3.1.

In the first step, which can be seen in Listing 4.8, the function transforms the input parameters into Hypergraphs, which is the data structure used as input for the GYO reduction.

Listing 4.8: Hypergraph Representation

```
// Create data structures to store relationships
vertex_to_edge := empty unordered_map of string to set of
   string
edge_to_vertex := empty unordered_map of string to set of
   string

// Read input data
for each kv in tablesAndPredicates
    edge := kv.first // edge of hypergraph is the table_name
    vertices := kv.second // vertices of hypergraph are the
       attributes

    // Associate vertices with hyperedges
    for each vertex in vertices
        equivalenceClass := edge + '.' + vertex //
           equivalenceClass is always table_alias.attribute
        if equivalenceClass exists in ds
            equivalenceClass := ds.find(equivalenceClass)
        vertex_to_edge[equivalenceClass].insert(edge)
        edge_to_vertex[edge].insert(equivalenceClass)
```

For efficiency reasons, an unordered map is used to store both directions of vertex and edge relationships of the hypergraph. These maps hold the name of a table as an edge, and the equivalence class representation of a column attribute as verta ex.

Once the input *tablesAndPredicates* is transformed into a hypergraph representation, the Flat-GYO algorithm is performed by applying the following three steps until either the vertex count reduces to one or no progress is made in a full iteration:

1. Eliminate all degree 1 vertices (i.e., vertices present in a single edge).
2. Remove empty edges.
3. Eliminate edges that are subsets of other edges.

Finally, the function will return a boolean value based on whether the count of remaining edges exceeds one. If true, it indicates that the GYO reduction was successful and a join tree could be constructed; otherwise, it returns false.

4.3.4 Reroot

In the case of 0MA queries all attributes involved in the aggregate function and the group by clause must originate from a single relation, the root node. A reroot operation

may be necessary due to the structure of the computed join tree. The reroot operation essentially reorganizes the tree by starting from the identified root node and adjusting the child-parent relationships accordingly.

Listing 4.9 shows the implementation of the reroot operation as pseudoce. It checks whether a node that requires rerooting exists. If such a node is found, the operation begins by selecting the identified root node. It then traverses the tree, inverting the child-parent relationships as it progresses. This process ensures that the root node becomes the ancestor of all other nodes in the tree.

Listing 4.9: Perform Rerooting

```
current := newRoot
while parents contains current: // newRoot is already root (has
    no parents)
    p := parents[current][0] // Each child has only one parent

    // remove current from child list of parent
    Remove current from join_tree[p]

    // add parent as child of current
    Append p to join_tree[current]

    current := p
```

4.3.5 Bottom-up semi-join reduction

The core step of this optimizer is implemented as part of a function called `bottomUpSemiJoin` and is intended to perform a bottom-up traversal of a join tree, creating semi-join queries for each subtree. The pseudocode for this is given in Listing 4.10. This is the first step that is done as part of Yannakakis' algorithm. In the following an overview of what exactly happens as part of this procedure is given.

The input of `bottomUpSemiJoin` consists of all attributes that were extracted in the previous steps, specifically the join tree that was computed using the GYO reduction described in 4.3.3, and other information about the input query that was described in 4.3.1. Also, the original select query represented as AST is used.

Listing 4.10: Bottom-Up Semi-Join

```
function bottomUpSemiJoin(select, join_tree,
   tableWithPredicateNames, tableObjects, predicateObjects, ds,
    selectionObjects)

    if tables or predicates are insufficient
        return
```

```
5
6     Find root
7
8     subQuery = buildJoinTreeRec(root, tableObjects,
          predicateObjects, tablesAndPredicates, join_tree, ds,
          selectionObjects)
9
10    Update SELECT query with subQuery
```

The main goal of the function is to manipulate the AST structure of the original query according to the join tree. The steps that are done as part of this function are:

- Do some initial checks e.g. if the number of tables in the SELECT query are sufficient for processing. If not, it returns early.
- Find root node in join tree.
- Call recursive function `buildJoinTreeRec` for root node and do the following:
  - Construct AST subqueries in a recursive matter.
  - Process direct neighbors of the current root.
  - Join construction for each neighbor.

As one can see the main logic of the AST manipulation happens in `buildJoinTreeRec`. In the following, the main steps that are performed as part of this function will be explained. Listing 4.3.3 gives again the pseudocode for this and will be used as a reference.

Listing 4.11: Build Join Tree Recursively

```
1  function buildJoinTreeRec(rootIdentifier, tableObjects,
       predicateObjects, tablesAndPredicates, adjacent, ds,
       selectionObjects, subQueryCnt)
2
3      subqueryName = generateSubqueryName(subQueryCnt)
4      subquery = createSubqueryTemplate(subqueryName)
5
6      Add root table to subquery
7      Add selection predicates associated with root table to
           subquery
8
9      for all neighborIdentifier in adjacent[rootIdentifier]
10         if neighborIdentifier is a leaf node
11             neighbor = tableObjects[neighborIdentifier]
```

```
12              Add selection predicates associated with
                    neighborIdentifier to subquery
13          else
14              subQueryCnt = subQueryCnt + 1
15              neighbor = buildJoinTreeRec(neighborIdentifier,
                    tableObjects, predicateObjects,
                    tablesAndPredicates, adjacent, ds,
                    selectionObjects, subQueryCnt)
16          end if
17
18          joinPredicates = getJoinPredicates(rootIdentifier,
                neighborIdentifier, predicateObjects,
                tablesAndPredicates, ds)
19          join = createJoin(rootIdentifier, neighborIdentifier,
                joinPredicates)
20          Add join to subquery
21      end for
22
23      Configure subquery
24
25      Return subquery
26  end function
```

Recursive Subquery Construction (line 1-15)

When `buildJoinTreeRec` is called the first thing it will do is generate a unique name for the subquery based on the current `subQueryCnt`, which is a global counter that makes sure that every subquery has a unique identifier and is always incremented every time `buildJoinTreeRec` is called.

The generation of a subquery happens within a function called `makeSubqueryTemplate` which takes the unique name as an input parameter and returns an AST.

Next, all neighbors of the current root will be iterated. For every neighbor node, there can happen two things depending on whether the node is a leaf node in the join tree or not:

- Node is a leaf node: The corresponding ASTPtr to that node will be used and be set as a `TablesInSelectElement`.
- Node is not a leaf node: `buildJoinTreeRec` will be called again, this time with the neighbor as a root node. The returned ASTPtr to that subquery will be set as a `TablesInSelectElement`.

Join Construction (line 17-19)

For each neighbor, it creates a join AST node (`ASTTableJoin`) and configures it based on whether join predicates exist. The join predicates are identified using the disjoint set data structure described in Section 4.3.2.

If no join predicates exist, a cross-join is created. Otherwise, a left semi-join is constructed with the join predicates. The join node is then added to the right table's AST representation (rightTable).

Configure Subquery (line 23)

In the last step of `buildJoinTreeRec`, some additional modifications are applied to the subquery before it is ultimately returned. This includes setting a WHERE clause containing all filter predicates from the original query, that can be applied within the just-created subquery. Essentially, this process represents a form of selection pushdown, facilitating the execution of filtering operations at the earliest feasible stage.

4.4 Other Modifications to ClickHouse

In addition to the modifications described in the previous sections, a flag was introduced that allows one to turn on or off the optimizer during runtime. This flag is quite convenient for the process of comparing results and assessing performance.

To activate or deactivate the optimizer in ClickHouse on the fly, the SET command is used within a SQL query:

```
SET optimizer = 1;
```

In the code, an attribute is used to store the value of this setting and through a conditional statement, the optimizer is either enabled or disabled.

CHAPTER 5

Evaluation

In this chapter, we evaluate the optimizer presented in Chapter 4. We do this evaluation using a set of 0MA queries, running them on both the original ClickHouse system and the system with the implemented optimizer. The comparison focuses on two performance metrics: runtime and memory consumption. The corresponding setup including the queries used can also be found on GitHub [13].

Section 5.1 provides an overview of the test setup used for this evaluation. Sections 5.2 and Section 5.3 discuss the testdata and the queries selected for the evaluation, including the rationale behind their selection. In Section 5.5, we further define and elaborate on the performance metrics used. Finally, Section 5.6 presents the evaluation results.

5.1 Setup and Methodology

To evaluate the performance of the modified ClickHouse instance, a straightforward setup was used. An overview can be seen in Figure 5.1. At the core of this setup is a Jupyter notebook. Initially, all queries were loaded as strings, available in two variants: one with standard queries and another with the appended string `SET optimizer = 1;`, indicating that the optimizer should be applied.

ClickHouse offers a feature that allows one to select tables from different sources, including other database systems. During the evaluation phase, this functionality was utilized by connecting to a pre-configured PostgreSQL database containing the test data. This means in practice that ClickHouse would still be used as a database system, but the tables containing the test data would be loaded from a Postgres instance.

The decision to use PostgreSQL for this purpose was based on two main reasons. Firstly, while ClickHouse generally follows the ANSI SQL standard, there are some exceptions, particularly with the *CREATE* statement, where a table engine must be specified. This would have required modifying all statements for creating a test database. Secondly,

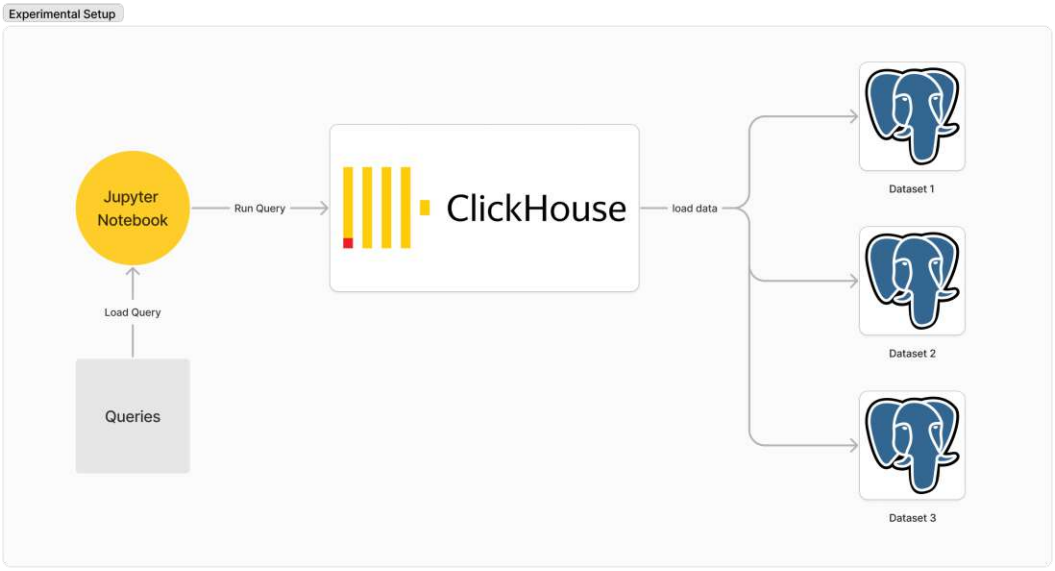


Figure 5.1: Setup High-Level Overview

PostgreSQL is widely used and there are many datasets available as pre-configured docker containers, which significantly simplifies the setup process.

To integrate PostgreSQL tables into the evaluation, the table part of each query was replaced with a statement that reads from the PostgreSQL instance:

```
SELECT * FROM
postgresql('localhost:5432', 'public', 'table', 'user', 'password');
```

Once the test queries were prepared, the ClickHouse Python client was used to execute these queries on a ClickHouse server instance.

The server instance was started by directly running the executable produced by the compiler, as we found that using an IDE such as CLion has a significant impact on the performance.

5.2 Testdata

In order to assess the performance of the modified ClickHouse system compared to the original system without the optimizer, an appropriate dataset was needed.

Given that our optimizer primarily focuses on evaluating join queries, particularly aiming to reduce intermediate results, our dataset needed to comprise numerous relations, ideally with each relation containing a substantial number of tuples.

After reviewing related literature and similar experiments, including [30] and [16], we identified the following datasets as fitting candidates for our testing:

- IMDB: The **Join Order Benchmark** (JOB) [35] is a benchmark first introduced in [35] for evaluating query optimizers. The dataset that is used for this benchmark comes from the **Internet Movie Database (IMDB)** [12]. It contains various information about movies, including actors, producers, and more. This benchmark transformed a May 2013 snapshot of the IMDB dataset, totaling 3.6 GB, into 21 distinct tables. Furthermore, the benchmark includes 33 queries, each with 2-6 variants, resulting in a total of 113 queries.
- SNAP: The **Stanford Network Analysis Platform** (SNAP) offers a collection of over 50 large network datasets [17]. These datasets cover diverse domains such as social networks, web graphs, and citation networks. For our evaluation, the used data is focused on patent relations, containing over 3 million different patents and their citations.

5.3 Queries

The evaluation was done using two sets of queries on the two previously mentioned datasets. In both cases, the queries required slight modifications to the FROM part of the query, as the data for our evaluation was stored in a PostgreSQL instance, and ClickHouse requires additional meta-information to identify the correct data source, as detailed in the setup Section 5.1.

For the first set of queries that were done on the IMDB dataset, a subset of queries was taken from the paper [30]. For the second dataset, which involves patent relationships from SNAP, the queries used are constructed in increasing path length, starting from length 2 and going up to length 5. If necessary, queries were also slightly adapted regarding the SELECT part in order to satisfy the properties of a 0MA query.

5.4 Hardware

The used hardware for running all evaluations was a MacBook Pro 16-inch using the M1 SoC with 32 GB of memory and a 512 GB SSD.

5.5 Performance Metrics and Evaluation Criteria

To measure the impact of our optimizer, we looked into the following two performance metrics:

- Runtime: This metric captures the time required for a query to execute, measured in seconds. Lower values indicate better performance.

- Memory consumption: This metric reflects the memory utilized during query processing, measured in Megabytes (MB). Reducing memory usage enhances system resource allocation, contributing to better performance. For the evaluation, we always looked at the peak memory consumption during the execution of a query.

It is important to note that a manual timeout of 50 seconds was configured for the evaluation, meaning any query execution exceeding this threshold would be automatically terminated by the system. Additionally, both query execution time and memory consumption can be significantly influenced by external factors, such as other processes running on the test system and CPU optimizations. To minimize these effects, all test queries were executed five times for both ClickHouse systems, the original one and the one using the optimizer.

5.6 Results

The runtime performance results are presented in Table 5.1, while the memory consumption results are shown in Table 5.2. Both tables report the average of the corresponding performance metrics (runtime and memory consumption) and the standard deviation ($\pm$) for the base system (without optimizer) and the modified system (with optimizer). The average and standard deviation are computed by running the evaluation five times in total for both performance metrics.

Additionally, a column is included that shows the difference between the two measurements, calculated as the optimized system minus the base system, meaning a negative value indicates an improvement. In case of a timeout, the corresponding entry is left blank.

The bar charts in Figure 5.2 and in Figure 5.3 summarize the reported numbers in the tables visually for both runtime and memory consumption.

In the following, the presented results will be discussed. First in Section 5.6.1 we present some general observations. In Section 5.6.2 we look in more detail into the snap queries, including the effects of disabling timeouts. Finally, in 5.6.3 we talk about the applicability of our optimizer.

5.6.1 General Observation

The evaluation results in Table 5.1 and Table 5.2 show the significant impact our optimizer can have on the ClickHouse system when it comes to the processing of certain types of join queries.

In the case of the IMDB dataset, one can see that queries where the original ClickHouse system had a timeout can now be solved in a couple of seconds (query 17.*). This is in line with the findings in [30] which also shows that for the chosen queries, conventional optimizers struggle to efficiently process them. Furthermore, our modified system has

Table 5.1: Runtime measured in seconds

Dataset	Query	Without Optimizer	With Optimizer	Difference Runtime
imdb	17a.sql	timeout	14.95 ± 2.07 s	-
imdb	17b.sql	timeout	15.20 ± 1.58 s	-
imdb	17c.sql	timeout	15.77 ± 2.45 s	-
imdb	17d.sql	timeout	15.07 ± 2.27 s	-
imdb	17e.sql	timeout	15.30 ± 1.97 s	-
imdb	20a.sql	22.57 ± 0.67 s	18.42 ± 2.55 s	-4.14181
imdb	20b.sql	19.59 ± 1.16 s	16.86 ± 2.18 s	-2.73661
imdb	3a.sql	5.19 ± 0.48 s	3.77 ± 0.60 s	-1.420542
imdb	3b.sql	3.52 ± 0.23 s	2.75 ± 0.53 s	-0.776138
imdb	3c.sql	6.77 ± 0.54 s	4.05 ± 0.57 s	-2.720183
imdb	q2a.sql	4.23 ± 0.35 s	3.45 ± 0.66 s	-0.776462
imdb	q2b.sql	4.35 ± 0.36 s	3.52 ± 0.34 s	-0.827537
imdb	q2c.sql	4.18 ± 0.18 s	3.09 ± 0.54 s	-1.097451
imdb	q2d.sql	5.01 ± 0.37 s	3.22 ± 0.47 s	-1.784314
imdb	q5a.sql	1.63 ± 0.13 s	2.49 ± 0.35 s	0.863208
imdb	q5b.sql	1.61 ± 0.29 s	1.61 ± 0.24 s	0.002995
imdb	q5c.sql	2.18 ± 0.12 s	2.78 ± 0.41 s	0.595921
snap	patents-path02.sql	17.48 ± 0.95 s	14.48 ± 1.03 s	-2.997295
snap	patents-path03.sql	37.37 ± 1.91 s	19.26 ± 2.11 s	-18.112656
snap	patents-path04.sql	timeout	26.40 ± 1.96 s	-
snap	patents-path05.sql	timeout	31.95 ± 1.64 s	-

significant enhancements in memory efficiency. All test queries show a significant decrease in memory consumption compared to the original system.

For the SNAP dataset and the chosen queries, similar observations can be made. All queries involving path lengths greater than three were not able to be solved by the original ClickHouse system. This aligns with the challenges faced by conventional optimizers, as discussed in [16], where even state-of-the-art systems struggle to efficiently evaluate these queries.

In the modified ClickHouse system, on the other hand, all queries were able to be solved within a reasonable time frame. The results also show that as the path length increases, the more advantageous our optimizer becomes regarding the runtime.

While the positive impact of the optimizer was clearly shown, some limitations still need to be addressed.

First, the integration did not always lead to an improvement, as can be seen in Table 5.1, where queries q5a, q5b, and q5c, have an increase in execution time with the optimizer enabled. Also, in some cases, the improvement can be seen as not significant e.g. 3b with a difference of under 0.1s. A possible explanation for this could be that the reduction in

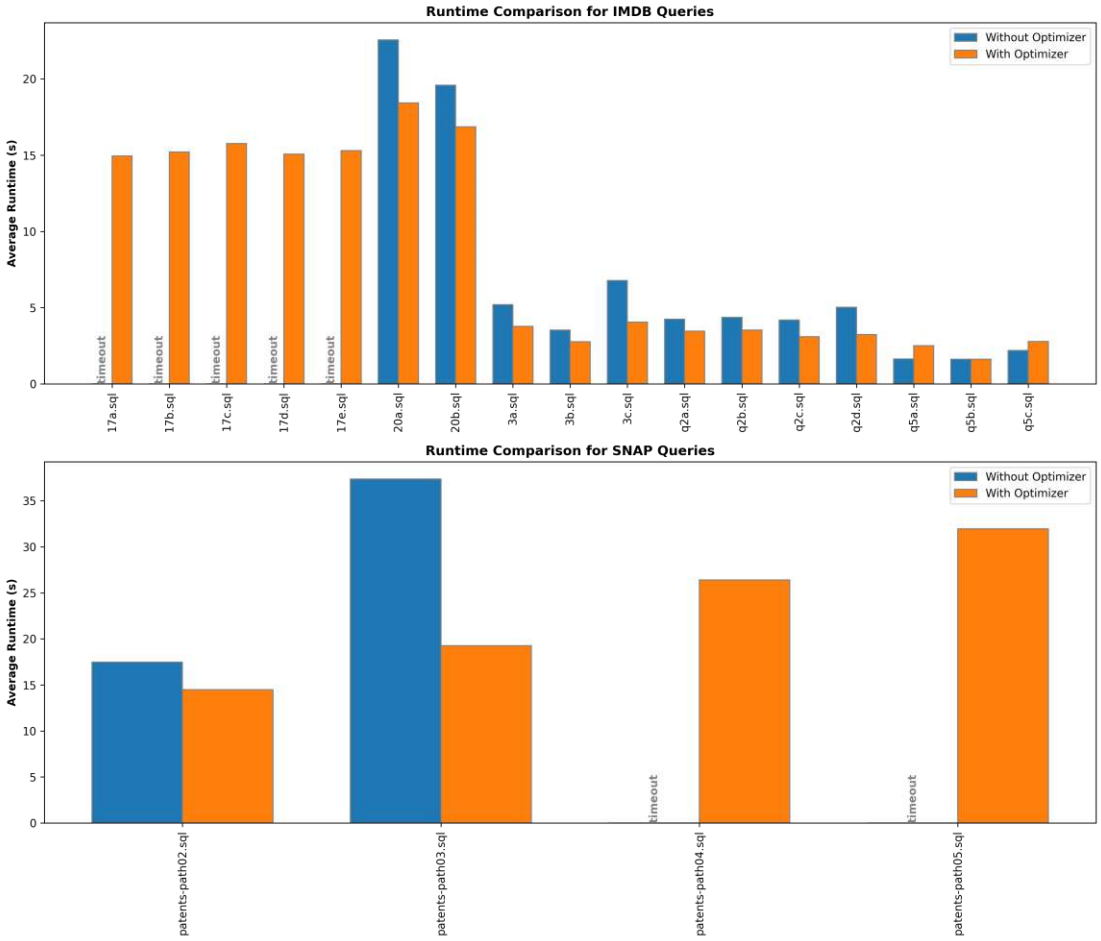


Figure 5.2: Runtime Comparison

execution time coming from the optimizer is smaller than the overhead introduced by the semi-join rewriting. It is important to note that this analysis is restricted due to the limited amount of test queries and datasets. In order to draw valid conclusions regarding these performance differences, it is necessary to conduct a more thorough investigation encompassing a greater variety of datasets and queries.

Secondly, the evaluation was performed on a single machine and with a specific set of queries. Also, all measurements were conducted with the experimental analyzer in ClickHouse disabled. Further testing is needed to validate the optimizer's effectiveness under various conditions, including different query types, larger datasets, and diverse deployment environments.

Table 5.2: Peak Memory consumption measured in MB

Dataset	Query	Without Optimizer	With Optimizer	Difference Memory
imdb	17a.sql	timeout	6719 ± 325 MB	-
imdb	17b.sql	timeout	6730 ± 381 MB	-
imdb	17c.sql	timeout	6458 ± 252 MB	-
imdb	17d.sql	timeout	6542 ± 317 MB	-
imdb	17e.sql	timeout	6458 ± 85 MB	-
imdb	20a.sql	9011 ± 556.71 MB	7190 ± 1026 MB	-1821 MB
imdb	20b.sql	7437 ± 1026.86 MB	6716 ± 323 MB	-721 MB
imdb	3a.sql	6641 ± 259.83 MB	6492 ± 294 MB	-149 MB
imdb	3b.sql	9393 ± 435.52 MB	6403 ± 425 MB	-2991 MB
imdb	3c.sql	6535 ± 760.14 MB	6477 ± 256 MB	-58 MB
imdb	q2a.sql	7528 ± 889.78 MB	6464 ± 93 MB	-1064 MB
imdb	q2b.sql	7810 ± 1057.93 MB	6275 ± 197 MB	-1535 MB
imdb	q2c.sql	9327 ± 514.49 MB	6882 ± 340 MB	-2445 MB
imdb	q2d.sql	8062 ± 401.19 MB	6990 ± 1522 MB	-1072 MB
imdb	q5a.sql	6593 ± 494.26 MB	6076 ± 216 MB	-516 MB
imdb	q5b.sql	9085 ± 344.97 MB	6402 ± 417 MB	-2683 MB
imdb	q5c.sql	6751 ± 135.01 MB	6730 ± 381 MB	-20 MB
snap	patents-path02.sql	10339 ± 464.41 MB	6585 ± 392 MB	-3754 MB
snap	patents-path03.sql	8248 ± 635.96 MB	6487 ± 411 MB	-1760 MB
snap	patents-path04.sql	timeout	6542 ± 375 MB	-
snap	patents-path05.sql	timeout	6632 ± 388 MB	-

5.6.2 Detailed Analysis

Despite the limitations outlined in the previous section, such as conducting the evaluation on a single machine with a limited dataset and set of queries, the results achieved by our optimizer are highly promising.

In the initial evaluation, we set a timeout threshold of 50 seconds for all queries. To better understand the effect of the optimizer and how ClickHouse behaves for computationally intensive queries, we wanted to see what happens when we turn off the timeout.

Interestingly, when we re-ran the SNAP queries **without the timeout** and again with and without the optimizer, ClickHouse still was not able to successfully execute certain queries in the case of the optimizer being disabled.

The results of this run without a timeout can be seen in Table 5.3 and 5.4 and clearly demonstrate the effectiveness of the optimizer. Without it, ClickHouse struggled with high memory usage, leading to timeouts and system crashes in some cases. In contrast, the optimizer significantly reduced both runtime and memory consumption, enabling all queries to be completed successfully. Although it was possible to compute the results for patents-path04.sql without the optimizer this time, the execution time was extremely

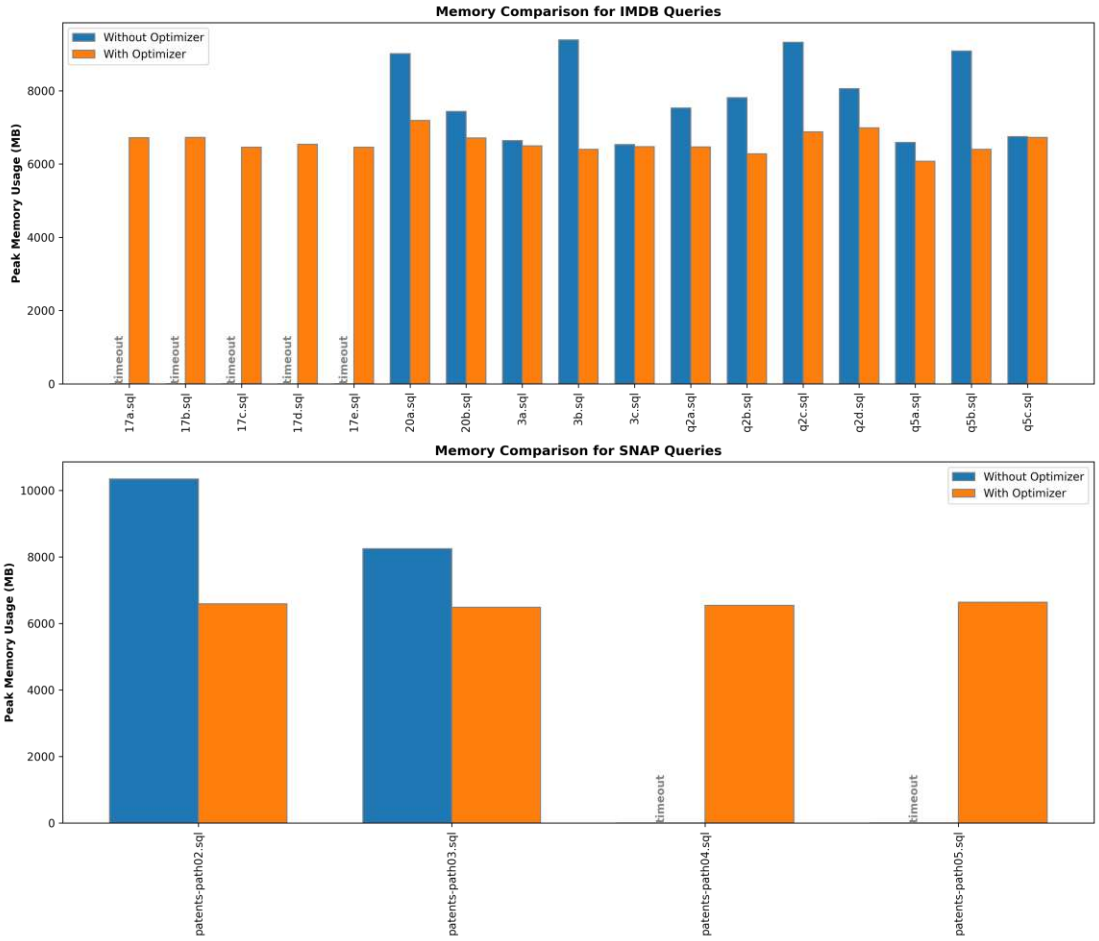


Figure 5.3: Peak Memory Consumption Comparison

Table 5.3: Runtime SNAP queries without timeout

Dataset	Query	Without Optimizer	With Optimizer	Difference Runtime
snap	patents-path02.sql	15.79 s	9.12 s	-6.67 s
snap	patents-path03.sql	36.42 s	14.10 s	-22.32 s
snap	patents-path04.sql	104.01 s	19.94 s	-84.07 s
snap	patents-path05.sql	-	24.53 s	-

high compared to using the optimizer. The improvements for this query were also the most significant ones, where runtime decreased by 84.07 seconds and memory usage dropped by 3900 MB.

The runtime comparison in Figure 5.4 illustrates that for the snap queries ClickHouse benefits from the optimizer in any scenario, and the more challenging the query becomes

Table 5.4: Memory consumption SNAP queries without timeout

Dataset	Query	Without Optimizer	With Optimizer	Difference Memory
snap	patents-path02.sql	3538 MB	2325 MB	-1213 MB
snap	patents-path03.sql	5339 MB	2519 MB	-2820 MB
snap	patents-path04.sql	6468 MB	2568 MB	-3900 MB
snap	patents-path05.sql	-	2568 MB	-

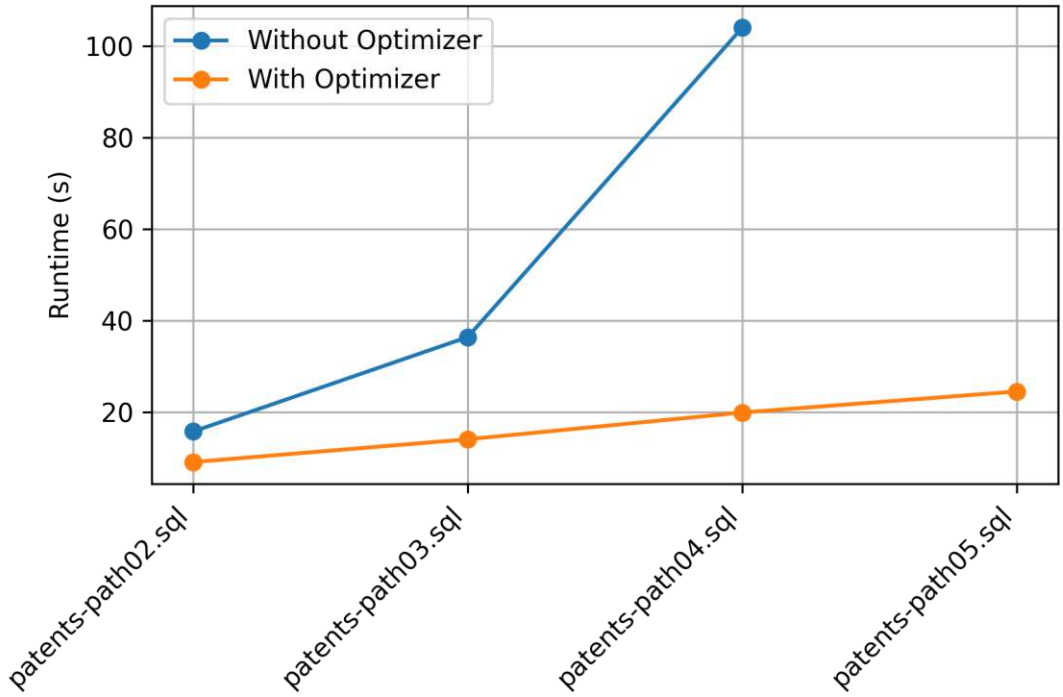


Figure 5.4: Runtime Comparison Snap Queries

due to increasing path length, the greater the benefit from the optimizer. The runtime for queries without the optimizer increased significantly with the path length, up to a point where the database instance was terminated. In contrast, the runtime with the optimizer enabled increased only marginally as the path length increased.

For memory consumption, which can be seen in Figure 5.5, a similar conclusion can be made. The memory usage was almost constant with the optimizer, while it increased dramatically without it. This consistency can be attributed to the optimizer's ability to efficiently remove overhead tuples, reducing the overall memory footprint.

With this in mind, we can conclude that the significant reduction in both runtime and memory usage coming from the optimizer shows the potential of structure-guided query optimization techniques. This improvement is in particular observable for computationally

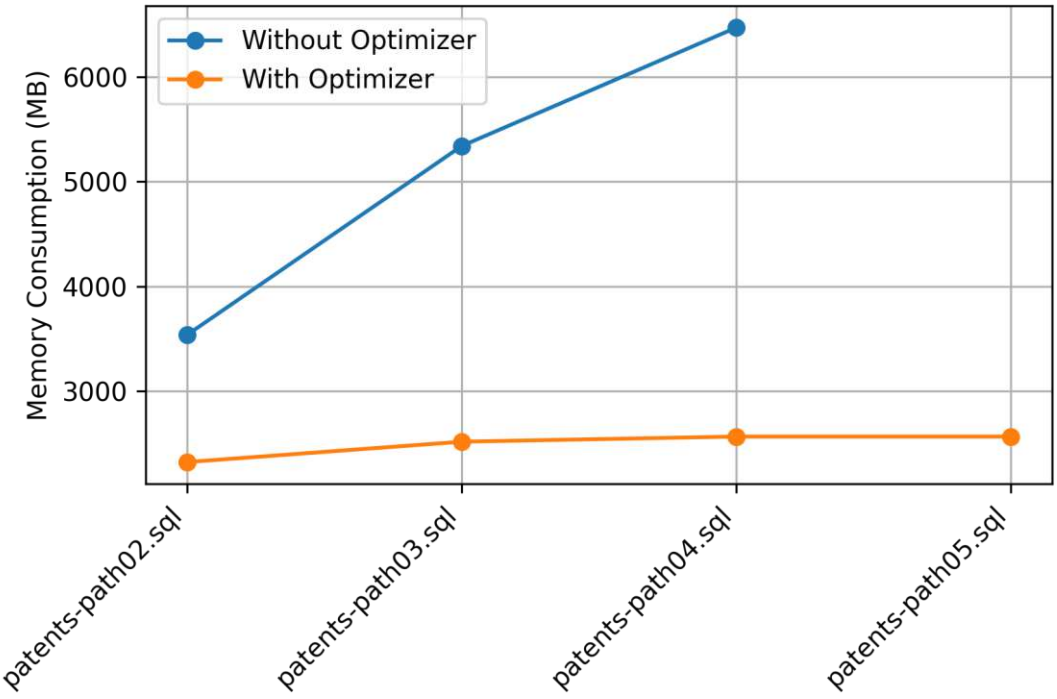


Figure 5.5: Memory Comparison Snap Queries

demanding queries, such as the SNAP queries, where the optimizer can be the difference between an unprocessable query due to intermediate result explosion and a query that executes efficiently, even within tight time constraints.

5.6.3 Applicability

Integrating a structure-guided query optimizer, as described in this work, into a column-store such as Clickhouse comes with certain challenges. The primary difficulty comes from the impedance mismatch with the Volcano Query Evaluation Model, which is commonly applied by many DBMS. Additionally, the current implementation of the optimizer is restricted to a limited subset of queries.

Despite this restriction, the optimizer has proven effective in executing complex queries that were previously infeasible. It must also be noted that the performance overhead introduced by the optimizer is minimal. The maximum observed overhead across all queries evaluated was only 4 milliseconds, with the majority of queries having less than 1 millisecond of overhead. In contrast, the optimizer has achieved significant reductions in execution time, with extreme cases showing reductions of up to 84.07 seconds. For these types of queries, it can therefore be said that the optimizer significantly reduces total runtime and significantly improves the overall performance of join query evaluations in ClickHouse.

These results align with recent research [1], [30] and suggest that integrating such optimizers into other database systems could be highly beneficial.

CHAPTER 6

Conclusion

In this thesis, we investigated a novel optimization technique: structure-guided query optimization, which leverages specific characteristics of join queries. Current research [31], [1] suggested that this technique could potentially mitigate intermediate result explosions and therefore improve the performance of join queries. However, it was unclear whether this approach could be implemented in a column-based database system and how it would perform in practice.

A key outcome of our investigation is that we demonstrated, for the first time to our knowledge, the feasibility of integrating a structure-guided query optimizer into a column-based database system.

Following the integration of the optimizer, we successfully showed that this approach is highly beneficial for certain types of queries (0MA), significantly improving runtime and reducing memory consumption. Additionally, we found that the optimizer enabled the execution of some queries that would otherwise fail due to intermediate result explosion.

Beyond the evaluation results that confirmed the performance improvements of structure-guided query optimization and its feasibility within a columnar system like ClickHouse, we also provided insights into the internal architecture and behavior of ClickHouse. These insights, necessary for the integration process, were not covered in existing documentation.

6.1 Open Questions

Currently, significant work is being done on an experimental analyzer in ClickHouse, as discussed in Section 4.2.3. An important open question for future research is how the optimizer will perform once the development of this experimental analyzer is completed, particularly with the introduction of the new abstraction level, the Query Tree. This new abstraction could potentially simplify and enhance the integration of a complete implementation of Yannakakis' algorithm.

Another open question concerns the optimizer's adaptability across different database systems and a wider range of queries. While in our evaluation we have seen that the optimizer shows great potential for one specific column-based system (ClickHouse) and one particular set of queries, its performance under varying conditions requires further research.

List of Figures

List of Tables

Bibliography

[1] [2406.17076] avoiding materialisation for guarded aggregate queries. https://arxiv.org/abs/2406.17076. (Accessed on 08/04/2024).

[2] ClickBench. https://benchmark.clickhouse.com/. [Accessed 1-Apr-2024].

[3] Clickhouse. https://clickhouse.com. [Accessed 13-April-2023].

[4] clickhouse-cloudflare. https://blog.cloudflare.com/http-analytics-for-6m-requests-per-second-using-clickhouse/. [Accessed 25-Feb-2024].

[5] clickhouse-distinctive-features. https://clickhouse.com/docs/en/about-us/distinctive-features. [Accessed 26-Feb-2024].

[6] clickhouse-github. https://github.com/ClickHouse/ClickHouse. [Accessed 11-March-2024].

[7] clickhouse-gitlab. https://www.youtube.com/watch?v=cMdQsxolcqc. [Accessed 25-Feb-2024].

[8] clickhouse-history. https://clickhouse.com/blog/introducing-click-house-inc. [Accessed 25-Feb-2024].

[9] clickhouse-table-engines. https://clickhouse.com/docs/en/engines/table-engines. [Accessed 2-March-2024].

[10] clickhouse-uber. https://www.uber.com/en-AT/blog/logging/. [Accessed 25-Feb-2024].

[11] db-engines. https://db-engines.com/en/ranking. [Accessed 25-Feb-2024].

[12] Imdb. https://www.imdb.com/. (Accessed on 05/04/2024).

[13] optimizer-benchmark. https://github.com/Jakob1010/query-optimizer-benchmark/tree/main. [Accessed 26-Feb-2024].

[14] optimizer-integration. https://github.com/Jakob1010/ClickHouse. [Accessed 26-Feb-2024].

[15] Show hn: A benchmark for analytical databases (snowflake, druid, redshift) | hacker news. https://news.ycombinator.com/item?id=32084571. (Accessed on 04/01/2024).

[16] Spark eval github. https://github.com/arselzer/spark-eval. (Accessed on 04/01/2024).

[17] Stanford large network dataset collection. https://snap.stanford.edu/data/. (Accessed on 05/04/2024).

[18] Vldb 10 years awards. https://vldb.org/archives/10year.html. (Accessed on 10/27/2023).

[19] why-clickhouse-is-so-fast. https://clickhouse.com/docs/en/concepts/why-clickhouse-is-so-fast. [Accessed 26-Feb-2024].

[20] why-clickhouse-is-so-fast-presentation. https://presentations.clickhouse.com/meetup59/building_for_fast/. [Accessed 26-Feb-2024].

[21] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreaos, and S. Madden. *The Design and Implementation of Modern Column-Oriented Database Systems*. 2013.

[22] D. J. Abadi. Column stores for wide and sparse data. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 292–297. www.cidrdb.org, 2007.

[23] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 967–980, New York, NY, USA, 2008. Association for Computing Machinery.

[24] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 411–422. VLDB Endowment, 2007.

[25] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999.

[26] J. Dittrich, J. Nix, and C. Schön. The next 50 years in database indexing or: The case for automatically generated index structures. *Proc. VLDB Endow.*, 15(3):527–540, 2021.

[27] Elmasri and Navathe. Fundamentals of database systems. 2017.

[28] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[29] L. Ghionna, L. Granata, G. Greco, and F. Scarcello. Hypertree decompositions for query optimization. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 36–45, 2007.

[30] G. Gottlob, M. Lanzinger, D. M. Longo, C. Okulmus, R. Pichler, and A. Selzer. Structure-guided query evaluation: Towards bridging the gap from theory to practice, 2023.

[31] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[32] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, jun 1993.

[33] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[34] B. Imasheva, N. Azamat, A. Sidelkovskiy, and A. Sidelkovskaya. The practice of moving to big data on the case of the nosql database, clickhouse. In H. A. Le Thi, H. M. Le, and T. Pham Dinh, editors, *Optimization of Complex Systems: Theory, Models, Algorithms and Applications*, pages 820–828, Cham, 2020. Springer International Publishing.

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

[36] D. Maier, J. D. Ullman, and M. Y. Vardi. On the foundations of the universal relation model. *ACM Trans. Database Syst.*, 9(2):283–308, jun 1984.

[37] T. Neumann and B. Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 677–692, New York, NY, USA, 2018. Association for Computing Machinery.

[38] R. Pichler. Database theory. Course materials and lecture notes, 2022. Course materials and lectures.

[39] R. Ramakrishnan and J. Gehrke. Database management systems (3. ed.). 2003.

[40] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: Not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, aug 2008.

[41] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 553–564. VLDB Endowment, 2005.

[42] M.-E. Vasile, G. Avolio, and I. Soloviev. Evaluating influxdb and clickhouse database technologies for improvements of the atlas operational monitoring data archiving. *Journal of Physics: Conference Series*, 1525:012027, 04 2020.

[43] A. Wickramasekara, M. Liyanage, and U. Kumarasinghe. A comparative study between the capabilities of mysql and clickhouse in low-performance linux environment. In *2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer)*, pages 276–277, 2020.

[44] M. 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.

[45] C. Yu and M. 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.

[46] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.