Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master's Thesis

Dynamically Exploiting Factorized Representations

Author: Paul Groß (2776170)

1st supervisor: Prof. Dr. Peter Boncz
daily supervisor: Daniël ten Wolde
2nd reader: Dr. Pedro Holanda

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 17, 2024

Abstract

While factorization and Worst-Case Optimal Join (WCOJ) algorithms promise significant performance improvements, their widespread adoption lacks behind because they are complex to implement and still face query optimization challenges. We aim to address this problem by proposing an adaptive, lightweight solution. We integrate factorized representations into DuckDB using pointers to hash table chains of a *Linear-Chained* hash table. This newly proposed hash table combines linear probing with chaining to insert a tuple only in a collision chain when their keys match, enabling collision-free chains. We use d-representations to (1) efficiently calculate aggregates and (2) perform cyclic joins in a worst-case optimal manner. By operating on d-representations, we avoid the explosion of intermediate results and efficiently reuse cached results, achieving speedups of up to 17.58x for aggregate computations and 16.77x for cyclic joins. To ensure that the new techniques are only employed when beneficial, we propose *adaptive factorization*, shifting the decision to use factorization from the planning stage to runtime. We then can collect statistics, which allows for accurate decision-making even in sub-queries or quering Parquet files. The metrics are then used by machine learning models to predict whether factorization would be beneficial. These models demonstrate an accuracy of 88% in our benchmarks.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

BI Business Intelligence. 37, 46, 48, 49

CSR compressed sparse row. 34

CSV comma-separated values. 33

DBMS database management system. 1, 3, 5, 6, 10, 14, 17, 30, 32, 33, 35, 48

FK foreign key. 9

GDBMS graph database management system. 1, 3, 6, 7, 22, 23, 25, 26, 34–36, 48

GQL Graph Query Language. 24

HJ hash join. 17

HLL HyperLogLog. 102

HT hash table. 17, 54

INLJ index nested loop join. 14, 15

JAO Join Attribute Ordering. 27, 46

JSON JavaScript Object Notation. 33

LP Linear probing. 70

NLJ nested loop join. 15, 17

NUMA non-uniform memory access. 14, 32, 33

OLAP online analytical processing. 1, 6, 10, 50

OLTP online transaction processing. 1, 6

PK primary key. 9

RDBMS relational database management system. 7, 10, 33, 36, 46

SIMD single instruction, multiple data. 32, 39

SMJ sort merge join. 16

SQL Structured Query Language. 7, 10, 11, 24, 25

SQL/PGQ Property Graph Queries. 24, 25, 34

TPC Transaction Processing Performance Council. 70

TPC-DS Transaction Processing Performance Council - Decision Support. vii, ix, 70, 74, 76, 77, 111

TPC-H Transaction Processing Performance Council - H. vii, ix, 70–72, 74, 75, 111

WCOJ Worst-Case Optimal Join. ii, ix, 2, 3, 23, 26–29, 34–39, 42, 46, 48–51, 92–95, 109, 111, 112

1 Introduction

In the era of big data, efficient data storage, retrieval, and management have become critical components for the performance of modern applications [51]. Database management systems (DBMSs) are the backbone for this era by handling vast amounts of structured and unstructured data across various industries, from finance and healthcare to social media and e-commerce [85].

While DBMSs traditionally run on their own servers with multiple connected clients, embedded databases, such as SQLite, can handle data efficiently on an end user's device. These systems are widely adopted, with SQLite managing a trillion active databases on client devices [46]. While in-process systems for online transaction processing (OLTP) workloads are common, there was a gap for in-process online analytical processing (OLAP) systems [84], which are optimized for analytical, read-heavy workloads.

In-process OLAP systems like DuckDB [84] have gained large popularity, e.g., with the python package of DuckDB reaching 3,736,093 downloads in the May of 2024 [83]. This popularity also stems from the fact that most professionals do not actually deal with big data. Instead, modern data management is more concerned with efficiently handling smaller, more practical datasets, where systems running on end users' devices often outperform big, complex, distributed systems designed for massive data [95].

A specialized subtype of DBMSs, are graph database management systems (GDBMSs) like Neo4J [74] and Amazon Neptune [16]. These are explicitly designed for graph workloads, which differ from default OLAP workloads; for instance, GDBMSs often need to compute large, complex many-to-many (cyclic) joins. GDBMSs can be used for different (analytical) applications, including fraud detection and recommendations in finance, e-commerce, and social networks [89].

Recent research has introduced two novel techniques for efficiently handling graph workloads: (1) WCOJ algorithms, which can efficiently find cyclic patterns, and (2) factorized representations, which are a method of lossless compressing data on relationships, allowing for more efficient data representation and processing. While both WCOJ algorithms and factorized representations promise significant performance improvements, only a few systems have already adopted them. This leads us to the following research questions:

1. What approaches and implementations exist for WCOJ algorithms and factorized representations?
2. What are the current challenges in adopting WCOJ algorithms and factorized representations, especially for general-purpose systems?
3. How can we integrate WCOJ and factorized implementations into these systems?
4. How can we determine when and how to incorporate these technologies into query plans to ensure they are used only when they provide a clear benefit?

1.1 Contributions

Our main contributions are as follows:

- **Literature Review:** We perform a literature review to analyze which systems currently implement factorization and WCOJs and what hinders the widespread adoption of these technologies.
- **Linear-Chained Hash Table:** We introduce, implement, and evaluate *linear-chained* hash tables that combine linear probing with chaining for collision-free hash table chains and feature micro bloom filters, which will be part of DuckDB 1.1.
- **D-Representations in DuckDB:** We introduce factorization to the general-purpose system DuckDB by using pointers to collision-free hash table chains as factorized d-representations.
- **Factorized Execution:** We utilize d-representations to compute aggregations and cyclic joins more efficiently by reducing data expansion and reusing cached results.
- **Adaptive Factorization with Machine Learning:** We implement an adaptive factorization method using machine learning to decide when to apply factorization based on runtime conditions.

1.2 Outline

First, we provide the relevant background information for this thesis in Chapter 2. Section 2.1 contains an introduction to DBMS, how and in which form these systems store data, and how this data can then be accessed and modified by the end user. We then go into the internal database systems and discuss how they execute queries. One of these operators systems use for this is the `Join` operator, which will be introduced in Section 2.2. As this work aims to introduce Factorization and WCOJ into DuckDB, which are especially relevant in the efficient analysis of graphs, we will give an introduction to the field of GDBMS in Section 2.3, which also contains background information on the to-be-implemented techniques. For the implementation, relevant optimization strategies adopted by analytical DBMSs are listed in Section 2.4. Finally, we conclude our background introduction with an introduction to DuckDB and DuckPGQ (Section 2.5).

Following the presentation of the necessary background information, Chapter 3 provides an overview of the current state of research through a literature review. This section covers the latest developments in the adoption of both WCOJ (Section 3.1) and factorization techniques (Section 3.2). We then conclude with a summary of our findings and discuss the insights that can be applied to the current project in Section 3.3.

In Chapter 4, we display our main contributions. The key idea of using hash table chains for factorized representations can be found in Section 4.1. From this, we propose a modification of the hash table for hash joins (Section 4.2 & Section 4.3) and implement it into DuckDB (Section 4.4 & Section 4.5). We then evaluate this new approach in Section 4.6. Using the modified hash table, we discuss how we can use the hash table chains as factorized representations (Section 4.7) and how we can compute aggregates (Section 4.8.1) and cyclic joins (Section 4.8.2) efficiently on these representations.

In Chapter 5, we then propose our concept of *adaptive factorization*, which uses factorization only when beneficial by run-time strategy switching (Section 5.1). We evaluate this approach empirically (Section 5.2) and analyze metrics that can be used to make this runtime switching in Section 5.3. We then propose and evaluate our approach of using machine learning to make this runtime decision in Section 5.4.

Finally, we conclude this thesis with a discussion and list loose ends and potential opportunities for future work in Chapter 6. A final conclusion can be found in Chapter 7.

2

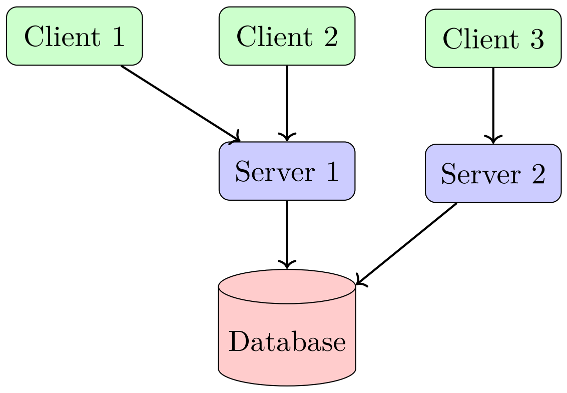
Background and Related Work

This chapter provides an introduction to database systems, explaining their purpose and the various types of databases that exist. Furthermore, this chapter will show how data in DBMSs can be manipulated and queried using query languages and shows how queries are translated into relational algebra and then into optimized query plans. It also covers the structure and normalized format of the data stored within these systems to motivate the importance of Join Operators. In this work, we aim to modify and improve the Hash Join Operator introduced in section Section 2.2.3 to use it for factorization and worst-case optimal joins. These two novel techniques will be introduced in Section 2.3.4 and Section 2.3.3. Both are optimization techniques that are primarily targeted for Graph workloads. We will further list how graphs can be processed by DBMSs and what the specific characteristics of graph workloads are.

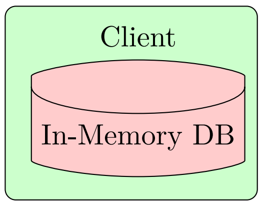
The chapter also addresses optimization techniques for analytical systems' read-heavy workloads, such as columnar storage, morsel-driven parallelism, and vectorized execution. Finally, we introduce DuckDB, this thesis's primary database system. It will later be used to implement and evaluate the proposed algorithms of this project.

2.1 Database Management Systems

A DBMS is a software system that stores, retrieves, and manages large sets of interrelated data efficiently and securely. The core functions of a DBMS include defining data structures, managing data manipulation, and ensuring data integrity and security against system failures or unauthorized access [91, p. 1]. An example application of DBMSs is in the context of banks, which can use the system to store and analyze information on their customers and their financial transactions. Traditional DBMSs typically run on a separate



(a) Database Server: standalone process, can handle multiple connections over a network.



(b) In-Process Database: Operates within the client application.

Figure 2.1: Comparison of a database server and an in-process database

server, handling requests from multiple clients. In contrast, there are embedded databases which run within the same process as the application. Such systems have gained immense popularity, with the most popular, SQLite, having over a trillion active databases in use [46]. These embedded or in-process databases can be used to manage data on the end user device (See Figure 2.1).

However, while there were in-process solutions for OLTP workloads like SQLite [46] for a long time, there was a gap for in-process OLAP systems [84]. Such systems are optimized for analytical, read-heavy workloads by, among other things, implementing a columnar storage format [1], a pipelined query execution engine [20], and morsel-driven parallelism [55]. For example, the embedded OLAP system DuckDB [84] has recently gained significant popularity, with its python package reaching 3,736,093 downloads in the May of 2024 [83].

Similar to OLTP or OLAP, another dimension to categorize DBMSs is by distinguishing between relational and non-relational databases. Non-relational databases are designed for handling unstructured or semi-structured data. They typically do not rely on predefined schemas and are designed to scale easily, making them suitable for managing large volumes of diverse data types. These databases typically support a variety of data models, including key-value or document stores [73]. Examples of such databases include MongoDB (storing JSON) [70] and Cassandra (a key-value store) [10; 4].

Another type of (typical) non-relational database are GDBMSs. These are optimized for high-performance graph querying and often feature special operators and a graph query language for efficient graph manipulation and analysis. These systems will be described in more detail in Section 2.3. Examples for GDBMSs are systems like Neo4J [74], Amazon Neptune [16], Kùzu [34], and TigerGraph [31].

In contrast, relational databases, such as SQLite and DuckDB, are relational model, which will be discussed in the next section.

2.1.1 Relational Database Systems

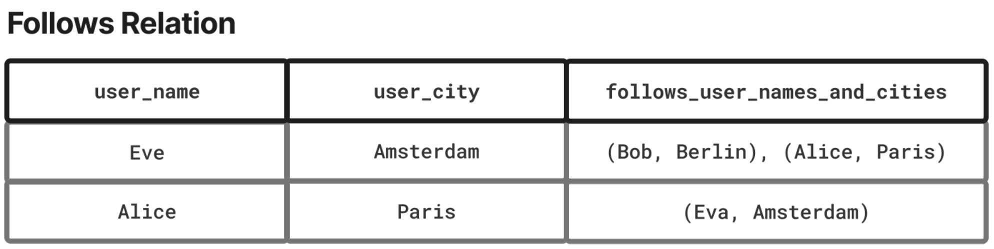
Introduced by Codd's 1970 paper, "A Relational Model of Data for Large Shared Data Banks" [26], relational database management systems (RDBMSs) have become the predominant type of database system. As of 2024, the top four most widely used databases are relational [30]. These systems use relational algebra to operate on data organized into tables (sometimes also called relations). Each table consists of rows (records) and columns (attributes), where each row represents a unique data item, and each column represents a data field. The power of relational databases lies in their ability to handle vast amounts of structured data and complex queries efficiently using the Structured Query Language (SQL) [91, pp. 12-15].

Normalization, first proposed by Codd as part of his relational model [26], is a design approach for relational databases to avoid the redundant storage of information, which can lead to inconsistencies and anomalies. Normalization aims to eliminate undesirable dependencies during insertion, update, and deletion, reduce the need for restructuring as new data types are introduced, make the relational model more user-friendly, and ensure that the database design remains efficient despite changes in query patterns [27].

The importance of normalization can be demonstrated using the example in Figure 2.2b, which illustrates a following relationship that could represent the underlying data of a social media application. The completely normalized relation includes both the users who follow each other and the city each user resides in for every row. However, the city a user lives in is functionally dependent on the user. For instance, Eve's city, Amsterdam, appears repeatedly whenever Eve follows someone. This redundancy can lead to anomalies during data insertion, update, or deletion: If one needs to update Eve's city of residence, it requires modifying all rows where Eve is following another user or being followed.

There are several levels of normalization, each with its own set of guidelines, known as normal forms [27; 104]:

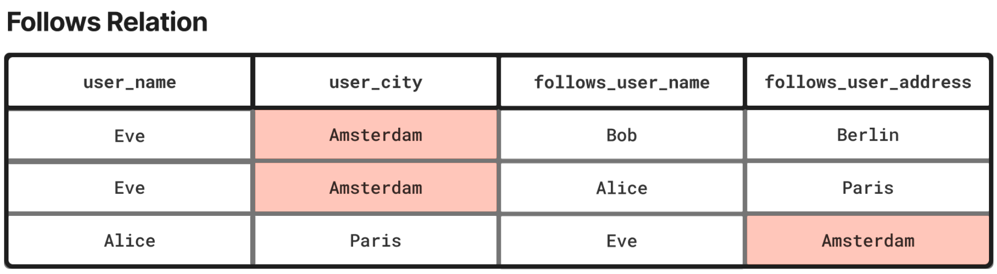
1. **First Normal Form (1NF)**: A table is in First Normal Form if it contains only atomic (indivisible) values, and each column contains values of a single type. This form eliminates repeating groups and ensures that the entries in a column are



Follows Relation

user_name	user_city	follows_user_names_and_cities
Eve	Amsterdam	(Bob, Berlin), (Alice, Paris)
Alice	Paris	(Eva, Amsterdam)

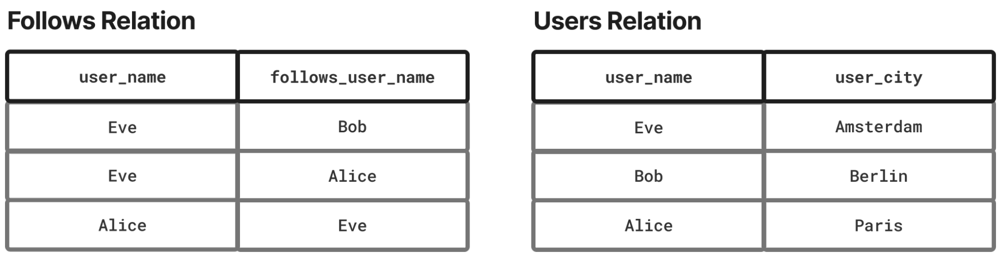
(a) Unnormalized relation for a follows relationship in social media. The first tuple (Eve, Amsterdam, ((Bob, Berlin), (Alice, Paris)) indicates that Eve, living in Amsterdam, follows Bob, who is based in Berlin and Alice, living in Paris.



Follows Relation

user_name	user_city	follows_user_name	follows_user_address
Eve	Amsterdam	Bob	Berlin
Eve	Amsterdam	Alice	Paris
Alice	Paris	Eve	Amsterdam

(b) First Normal Form (1NF) applied to the follows relationship. The nested tuple in the first row of the initial data is now expanded into two rows, one row for each user Alice is following. This leads to Amsterdam being present three times in the table.



Follows Relation

user_name	follows_user_name
Eve	Bob
Eve	Alice
Alice	Eve

Users Relation

user_name	user_city
Eve	Amsterdam
Bob	Berlin
Alice	Paris

(c) Second Normal Form (2NF) applied to the follows relationship. As the city of residence is dependent on the user (e.g., Eve $\rightarrow$ Amsterdam), the city relation is created as a new key, where the user_name acts as a key to connect the two relations.

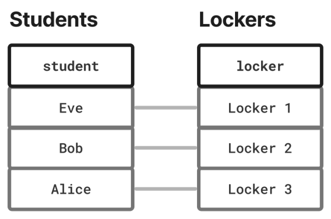
Figure 2.2: Normalization process for a follows relationship in social media, showing the initial unnormalized relation, application of 1NF, and 2NF.

of the same data type. In the example of Figure 2.2, the nested values of the `follows_user_names_and_cities` column are split into two columns, and the list of two values in the first-row result in respectively two separate rows.

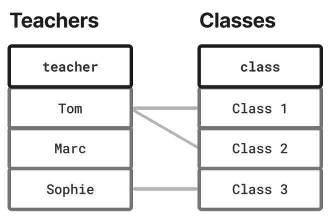
2. **Second Normal Form (2NF)**: A table is in Second Normal Form (2NF) if it is in 1NF and all non-key attributes are fully dependent on the entire primary key. For composite primary keys, each non-key attribute must depend on all parts of the key. 2NF eliminates partial dependencies, where non-key attributes depend on only part of a composite key. In the example of Figure 2.2, the primary key is a composite of `user_name` and `follows_user_name`. However, `user_city` and `follows_user_city` depend only on `user_name` and `follows_user_name` respectively, not on the entire composite key. Therefore, they are placed in separate tables.

3. **Third Normal Form (3NF)**: A table is in Third Normal Form if it is in 2NF and all its attributes are functionally dependent only on the primary key. There should be no transitive dependencies where non-key attributes depend on other non-key attributes. 3NF aims to reduce redundancy by ensuring that non-key attributes do not introduce additional dependencies beyond the primary key.

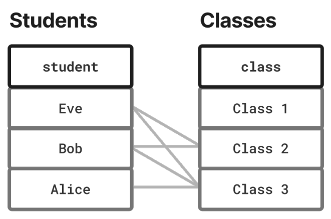
Normalized relations are then again linked together through primary key (PK) and foreign key (FK) keys. A PK uniquely identifies a record in a table, ensuring that each entry is distinct and can be efficiently retrieved. A FK is a field in one table that connects to a PK in another table, creating a relationship between the two tables [91, p.45]. These relations are distinguished based on their mapping cardinality as depicted in Figure 2.3.



(a) One-to-One: each student has one locker



(b) One-to-Many: one teacher can teach many classes



(c) Many-to-Many: many students can be in many classes

Figure 2.3: Different Types of Relationship Cardinalities

In the example in Figure 2.2c, the `user_name` of the `Users` relation is the primary key and the `user_name` and the `follows_user_name` of the `Follows` relation are two foreign keys linking the users in a many-to-many relationship (Figure 2.3c), as one user can follow and be followed by many other users.

In a DBMS, linking is achieved through a join operation, which allows data retrieval from two different (normalized) relations through a cartesian product [28], where tuples are combined if they satisfy a given join condition [67]. As this operation is frequent and resource-intensive, it is important to feature highly optimized algorithms for this task to achieve good performance in DBMS. This is especially important in OLAP systems, where queries can have many large joins. For details on this can be found Section 2.2.

To read and manipulate the data stored in a RDBMS, users typically use a relational query language like SQL. It was developed at IBM in the early 1970s by Donald D. Chamberlin and Raymond F. Boyce [22] after they were introduced to the relational model by Edgar F. Codd [26].

When an RDBMS receives a relational query language statement, such as an SQL statement, it parses the statement into a tree of relational algebra operators. Relational algebra is a set of operations used to manipulate and query relations. It enables relations to be filtered, linked, aggregated, or otherwise modified to formulate queries to a database [99]. The relations the algebra operates on are homogeneous sets of tuples defined as

$$S = \{(s_{j1}, s_{j2}, \ldots, s_{jn}) \mid j \in 1, 2, \ldots, m\}$$

where m is interpreted as the number of rows in a relation, and n is the number of columns. For each column, all tuples have the same type. Different relational algebra operators are defined to manipulate these homogeneous bags of tuples [91]. Among these operators are:

- **Selection** σ: Selects rows that satisfy a given predicate. $\sigma_{\text{condition}}(R)$

  `Example:` $\sigma_{\text{age}>30}(\text{Employees})$ selects all employees older than 30.

  `SQL Equivalent: SELECT * FROM Employees WHERE age > 30`

- **Projection** π: Selects specified columns from a relation. $\pi_{\text{attributes}}(R)$

  `Example:` $\pi_{\text{name, age}}(\text{Employees})$ only returns the `name` and `age` columns.

  `SQL Equivalent: SELECT name, age FROM Employees`

- **Cartesian Product** $\times$: Combines tuples from two relations pairwise. $R \times S$

  `Example:` Employees $\times$ Departments returns the combinations of all employees with all departments.

  `SQL Equivalent: SELECT * FROM Employees CROSS JOIN Departments`

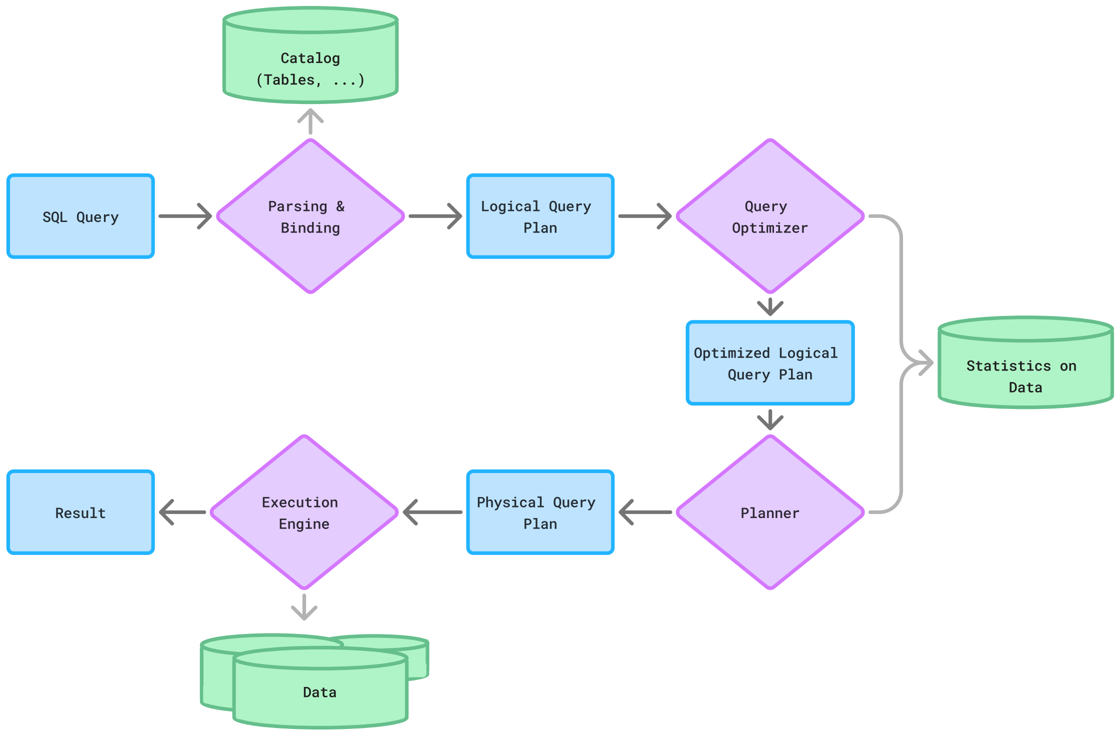


Figure 2.4: Steps in query execution

- **Join** $\bowtie$: Combines related tuples from two relations based on a condition. $R \bowtie_{\text{cond}} S$

  `Example:` Employees $\bowtie_{\text{Employees.dept\_id=Departments.dept\_id}}$ Departments only combines the tuples of both relations where the `Employees.dept_id` is equal to the `Departments.dept_id`.

  `SQL Equivalent: SELECT * FROM Employees JOIN Departments ON Employees.dept_id = Departments.dept_id`

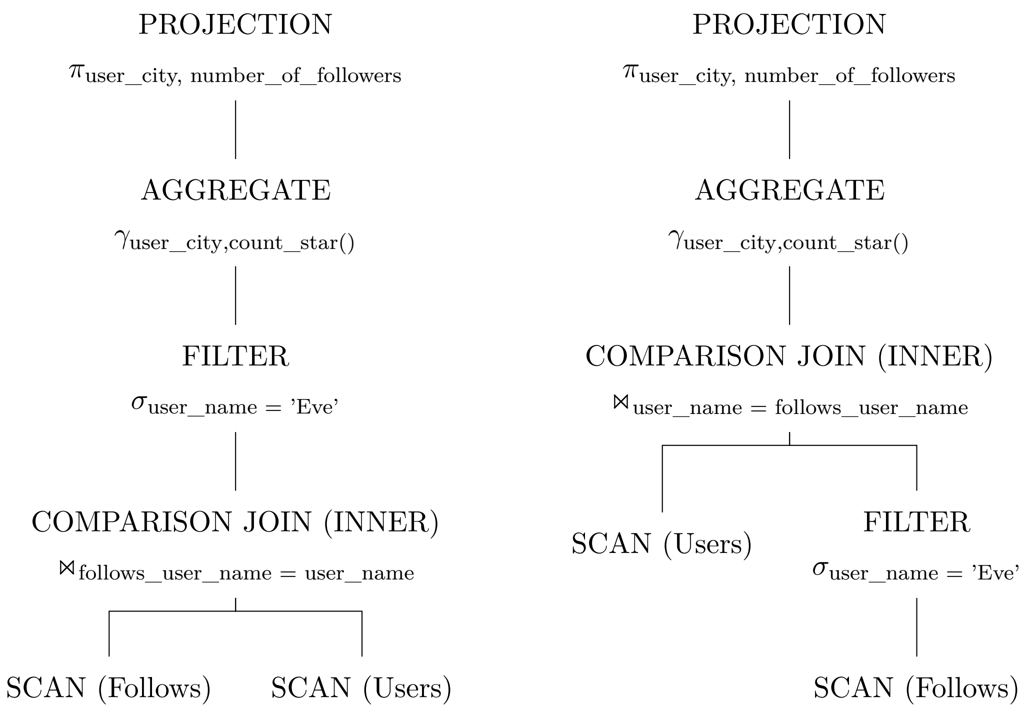
Translating the SQL syntax tree to the relational operator is an important step in processing a query, as discussed in the next section.

2.1.2 Query Processing

Query processing refers to the steps necessary to extract data from a database. These include translating high-level queries (like SQL) into expressions usable at the physical file system level, applying query optimization transformations, and executing the queries.

An overview of the query processing steps can be found in Figure 2.4. First, the system translates the SQL query into a parse tree, performing semantic checks by looking up table and column names in the catalog ("binding"), optimizing the resulting logical query plan, finally transforming it into a physical query plan, and then executing this plan [91, p. 527]. Typically, the tree is split into operator pipelines for execution that are run one after the other [42].

Each parsed SQL query can be translated into a logical query plan in multiple ways, providing different execution strategies for the same outcome. For example, given the query in Listing 4.4 where the number of followers per city for the user `Eve` is retrieved, directly translation the SQL syntax tree to relation algebra results in the tree in Figure 2.5a.


PROJECTION
$\pi_{\text{user\_city, number\_of\_followers}}$
AGGREGATE
$\gamma_{\text{user\_city,count\_star()}}$
FILTER
$\sigma_{\text{user\_name = 'Eve'}}$
COMPARISON JOIN (INNER)
$\bowtie_{\text{follows\_user\_name = user\_name}}$
SCAN (Follows)
SCAN (Users)
PROJECTION
$\pi_{\text{user\_city, number\_of\_followers}}$
AGGREGATE
$\gamma_{\text{user\_city,count\_star()}}$
COMPARISON JOIN (INNER)
$\bowtie_{\text{user\_name = follows\_user\_name}}$
SCAN (Users)
FILTER
$\sigma_{\text{user\_name = 'Eve'}}$
SCAN (Follows)

(a) Unoptimized Logical Plan **(b)** Optimized Logical Plan

Figure 2.5: Comparison of an unoptimized and Optimized Logical Plans

This unoptimized logical plan applies the filtering operation after joining the `Follows` and `Users` tables. This approach is correct but might not be the most efficient because the join operation, which is usually expensive, is performed on the entire dataset before filtering for the specific user `Eve`.

During the query optimization step, the plan is transformed to be more efficient to execute, resulting in Figure 2.5b. Here, the filter operation for the user `Eve` is pushed down before the join operation. This means that only the relevant subset of the `Follows` table (those rows where `user_name` = `Eve`) is considered during the join operation. Now, the join operation processes a smaller amount of data, which can lead to significant performance improvements. Furthermore, the position of the incoming relations into the join operator is swapped. Because of specific join implementations, having the smaller relation on the right is efficient. Since the number of users Eva follows will be smaller than the total number of Users in the system, using the filtered follows relation as the right side of the join is more performant. More details will be provided in Section 2.2.3. After the join, the aggregate operation is performed to count the followers for each city, and finally, the projection retrieves only the required columns (`user_city` and `number_of_followers`).

An optimized plan can greatly improve query performance, making the optimizer an important component. For example, using a user relation with 10,000 entries and a follower relation with 100,000 entries, using DuckDB with one thread results in a runtime of 24 milliseconds for the optimized plan and 134 milliseconds for the unoptimized plan. Through query optimization, the runtime was therefore reduced by 82.09%.

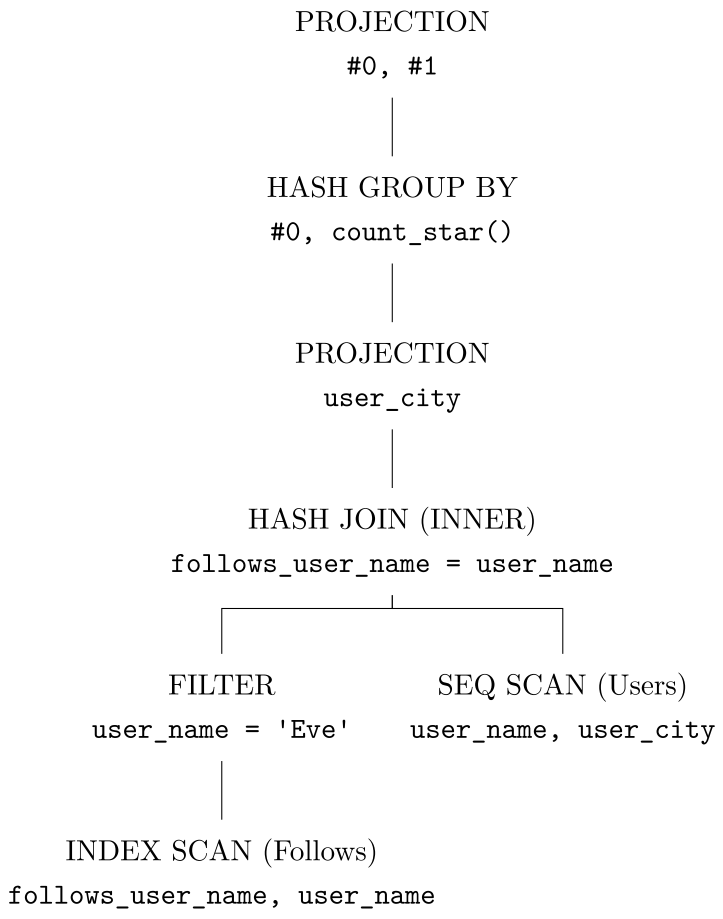


Figure 2.6: Physical Plan of the optimized logical plan in Figure 2.5b

The relational algebra representation of a query only partially specifies how to evaluate it, leaving room for various execution strategies. These strategies include different join methods, selection techniques, and indexing approaches, each affecting the efficiency and performance of query execution [91, p. 528]. The DBMS needs to transform the logical plan into a physical one that also defines how each logical operator is implemented and executed.

For the example logical plan in Figure 2.5b, the DuckDB physical plan is depicted in Figure 2.6. This plan illustrates how the logical operations are executed using specific physical operators. The projection operation is followed by a `HASH GROUP BY` operation to count the number of followers per city. DuckDB utilizes a hash table-based method known as `HASH JOIN` to execute the join operation. This is one of several join algorithms implemented by DuckDB; another example includes the index nested loop join (INLJ), which will be explained in Section 2.2.1. The condition `user_name = 'Eve'` is optimized by pushing the filter down as far as possible within the execution plan. The optimized results are subsequently joined using a sequential scan of the `Users` table and an index scan of the `Follows` table.

Since join operators will play a central role in this project, the next chapter will introduce different implementations of these operators.

2.2 Join Implementations for Relational Databases

A join can involve two relations (two-way) or more (multiway), depending on the number of relations they process at once. Typically, joining n relations involves $(n-1)$ two-way join [67], which are the types of joins this section will focus on. However, for cyclic multiway joins, it can be more efficient to use multiway worst-case optimal join algorithms [78], which will be discussed in Section 2.3.3.

Joins can be optimized across several aspects: From query optimization, which focuses on the join order and the decomposition of multiway joins into binary joins, to the use of parallel join processing that aims to use multiple cores or nodes to speed up execution [67]. For the project on hand, however, the primary focus is on implementing the actual join algorithm and leveraging optimizations discussed in Section 2.4, such as non-uniform memory access (NUMA) aware algorithm design and vectorized query execution.

The following sections will introduce three possible join algorithm implementations and compare them regarding their performance. The running example to explain the algorithms is to join the relation R on S under the condition $r(a)\ \theta\ s(b)$, which will result in relation

$Q = R \bowtie_{r(a)\ \theta\ s(b)} S$. S will be the outer table with a size m tuples which are stored on M memory pages. R will be the inner table with a size n tuples and N pages respectively.

2.2.1 Nested Loop Join

The nested loop join (NLJ) is the easiest way of implementing a join algorithm. In this method, one of the joined relations is chosen as the inner relation, and the other is chosen as the outer relation. For each tuple of the outer relation, all tuples of the inner relation are read and compared with the tuple from the outer relation. The two tuples are combined and emitted whenever the join condition is satisfied. The algorithm for performing the join $Q = R \bowtie_{r(a)\ \theta\ s(b)} S$ is defined as follows:

Algorithm 1 Nested Loop Join Algorithm

```
for each tuple $s \in S$ do
    for each tuple $r \in R$ do
        if $r(a)\ \theta\ s(b)$ then
            $Q \leftarrow Q \cup (r \times s)$
```

Here, R and S are the joined relations, r and s are tuples from R and S respectively, and θ is the join condition. This method, although simple, can be inefficient for large relations as it involves a quadratic number of comparisons.

The INLJ algorithm improves upon the nested loop join by using an index on the inner relation to speed up searches, significantly reducing the number of comparison operations required. This method is particularly effective when an index already exists for the join attribute of the inner relation [23, 43:21-46:46].

Algorithm 2 Index Nested Loop Join Algorithm

```
for each tuple $s \in S$ do
    for each tuple $r \in$ Index($r(a)\ \theta\ s(b)$) do
        $Q \leftarrow Q \cup (r \times s)$
```

In this method, R is typically chosen as the inner relation and is assumed to have an index on the joining attribute a. When a tuple s from the outer relation S is examined, the index on R is used to quickly find all matching tuples r that satisfy the join condition $r(a)\ \theta\ s(b)$. This direct access via the index avoids scanning the entire relation R, thus saving on I/O costs and making the join operation more efficient [91; 43].

The standard NLJ requires $n \times m$ comparisons, a comparison of every tuple in S against every tuple in R. For the index nested loop join, the number of comparisons depends on the selectivity of the index. It can be approximated as $m \times$ avg_matches per tuple, where avg_matches represents the average number of tuples in R that satisfy the join condition per tuple in S. Summarized, the cost for comparisons are as follows:

$$\text{Total Comparisons NLJ} : C_{\text{CMP, NLJ}} = C_{\text{CMP, BNLJ}} = n \times m \tag{2.1}$$

$$\text{Total Comparisons INLJ} : C_{\text{CMP, NLJ}} = m \times \text{avg\_matches} \tag{2.2}$$

2.2.2 Sort Merge Join

The sort merge join (SMJ) key concept is merging two sorted relations. For computing $Q = R \bowtie_{r(a)\ \theta\ s(b)} S$, both relations R and S are first sorted on the joining attributes $r(a)$ and $s(b)$ respectively. Although initially sorting the data can take a lot of effort, it makes the next step of combining matching entries from both groups much quicker, as only one pass over the sorted data is necessary:

Algorithm 3 Sort Merge Join Algorithm

```
sort $R$ on $r(a)$
sort $S$ on $s(b)$
$i \leftarrow 1$
$j \leftarrow 1$
while $i \le |R|$ and $j \le |S|$ do
    if $r_i(a) = s_j(b)$ then
        $Q \leftarrow Q \cup (r_i \times s_j)$
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    else if $r_i(a) < s_j(b)$ then
        $i \leftarrow i + 1$
    else
        $j \leftarrow j + 1$
```

This method involves scanning through the sorted lists of R and S. When matching tuples are found, they are combined and added to the result set Q. If a tuple from R is less than a tuple from S based on the join condition, the pointer for R is advanced. Conversely, if a tuple from R is greater, the pointer for S is advanced. This approach avoids the full Cartesian product typically required in nested loop joins and is significantly more efficient in terms of computational and I/O costs when dealing with large datasets. The efficiency of the sort-merge join is especially good when the data is already sorted or indices are available that can be utilized to speed up the sorting phase [91; 85].

The number of comparisons required in the merge sort algorithm is the sum of the comparisons needed for sorting the individual lists and the comparisons made while merging those lists. If quicksort is used for sorting the lists, the number of comparisons can be calculated as follows:

$$\text{Comparisons Sort R:} \quad C_{\text{CMP, SORT(R)}} = m \log m \tag{2.3}$$

$$\text{Comparisons Sort S:} \quad C_{\text{CMP, SORT(S)}} = n \log n \tag{2.4}$$

$$\text{Comparisons Merge S \& R:} \quad C_{\text{CMP, MERGE(R, S)}} = m + n \tag{2.5}$$

$$\text{Total Comparisons SMJ:} \quad C_{\text{CMP, SMJ}} = m(\log m + 1) + n(\log n + 1) \tag{2.6}$$

Assuming $n = m$, the algorithm therefore has a complexity of $\mathcal{O}(n \log n)$, which is much better than the NLJ with a complexity of $\mathcal{O}(n^2)$

2.2.3 Hash Joins

The last type of join implementation to discuss is the hash join (HJ) algorithm, which uses a hash table (HT) to perform the join operation. A hash table is a data structure that stores key-value pairs and enables fast retrieval by using keys to access values [63]. This is achieved using a hash function that maps keys of arbitrary size to fixed-size values [5].

In the context of DBMS, hash functions focus on efficiency and even hash distribution [23]. Examples of hash functions used in DBMS are xxHash [29], Google CityHash [40], or Google Farmhash [41].

The first step to create a hash table is to allocate a buffer of size N. A hash function is then defined to map keys to a hash within the range $[0, N]$. A common approach is to use a standard hash function like CityHash and then apply a modulo operation to ensure the hash falls within the desired range, see Equation (2.8).

$$h(k) = \text{hash}(k) \tag{2.7}$$

$$\text{where } h(k) \in [0, N-1]$$

$$\text{e.g. } h(k) = \text{CityHash}(k) \mod N \tag{2.8}$$

When inserting a key-value pair, the index into the buffer is calculated using the hash function, and the value is placed at the calculated index. For lookup operations, the index is similarly calculated on the lookup key using the hash function, allowing the program to directly access the value at the buffer location corresponding to the computed index [63].

Typically, hash tables such as the `std::unordered_map` in the C++ standard library also allow for the deletion of keys and can be resized [92]. However, in the context of database management systems, the required size of the hash table is often known in advance since the sizes of the relations to be joined are known. Additionally, values must only be inserted into the hash table and not removed.

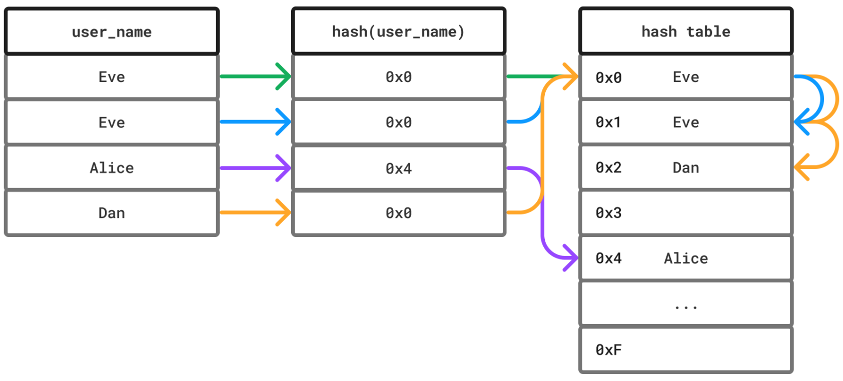
During insertion, there can be a case in which the buffer slot calculated using the hash is already filled. Such a collision can occur for two main reasons: duplicate keys or the hashing of different keys to the same index. For example, in a table where one user follows many others, and the hash table is constructed using `user_id` as the key, multiple entries might hash to the same index. Also, there might be a case where two different user names map to the same hash. This is called a hash collision [23].

In real-world applications, hash collisions are very common, as their probability follows the birthday paradox [93]: Given a hash function that produces a hash of b bits, the number of possible outputs is 2^b. However, in the case of hash tables, the number of distinct values of the hash is limited by the number of slots, which is much smaller than the value range of the hash. The probability of a hash collision for inserting n values into a hash table of size Ncan be calculated as.

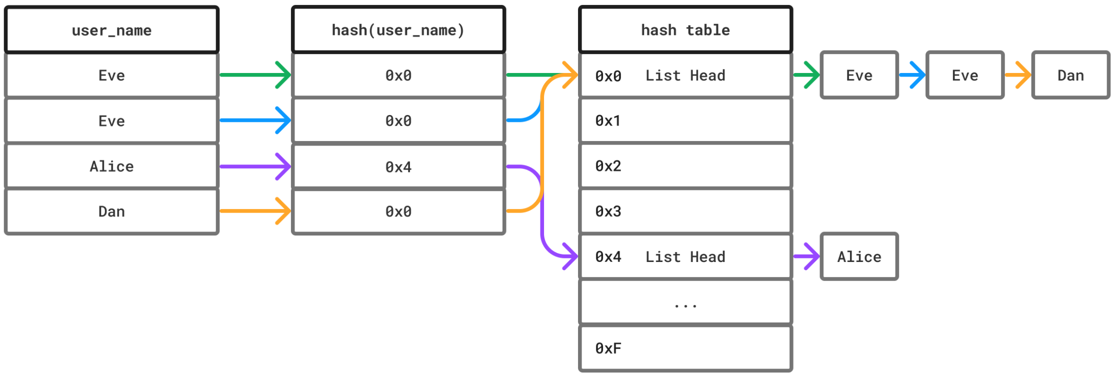
$$P = 1 - \prod_{i=1}^{n-1} \left(1 - \frac{i}{N}\right) \tag{2.9}$$

According to Equation (2.9), this means that for a hash table with 1024 slots, already for a fill rate of 5%, there will be a collision with a chance of 77%. This shows how important it is to find performant methods of dealing with collisions.

Two common methods to handle such collisions are linear probing and chaining. Both methods are depicted in Figure 2.7. Linear probing (Figure 2.7a) addresses collisions by moving sequentially through the buffer slots from the point of collision until an empty slot is found for the new key-value pair. This method is simple and efficient but can lead to clustering, where groups of adjacent slots get filled, slowing down the insertion and lookup process [63]. Clustering can be avoided by quadratic probing, where the interval between probes is increased quadratically (i.e. 1, 4, 9, 16, etc.), rather than linearly, filling the slots more randomly. However, this has the drawback of more cache misses in large-scale applications. While the next bucket is likely already in the cache in linear probing, large jumps will result in cache misses [88].



(a) Duplicate resolution using linear probing



(b) Duplicate resolution using chaining

Figure 2.7: Comparison between two methods of duplicate resolution in hash tables, linear probing, and chaining

On the other hand, chaining (Figure 2.7b) resolves collisions by storing all entries that hash to the same index in a linked list attached to each buffer slot [23]. While linear probing is categorized under open addressing, using linked lists to manage collisions is known as closed addressing.

Liu et al. [61] show that closed addressing is better for hash tables with high fill rate or skewed data, as the number of filled slots is smaller and therefore the number of collisions is reduced. This also leads to hash tables using closed addressing tend to be more stable in performance [61]. On the other side, open addressing like linear probing reduces the number of cache misses, as there are no list pointers that need to be followed, making them more performant for lower fill rates [23].

The hash join algorithm consists of two steps: First, a hash table is constructed on one of the relations, referred to as the build side. This step involves calculating the keys' hashes and inserting the key value parts into the hash table array. After the build, the second

Algorithm 4 Hash Join Algorithm with Linear Probing

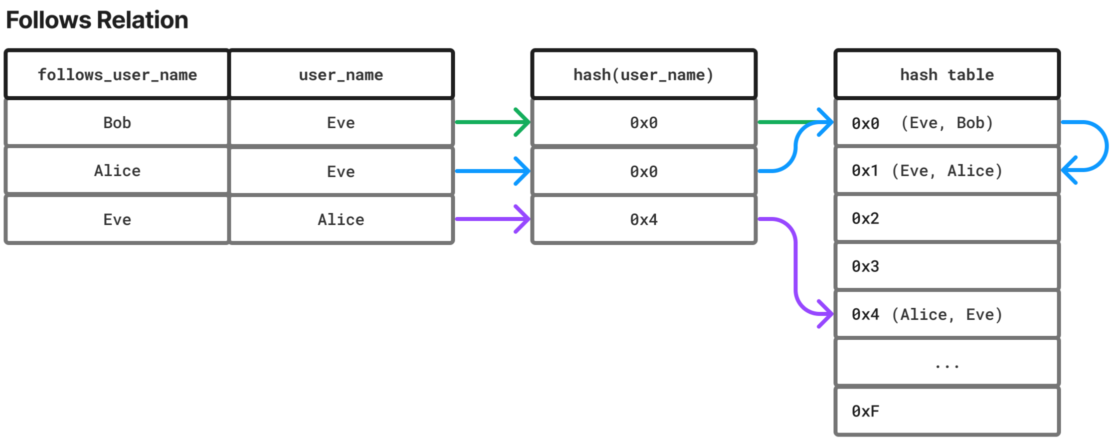
```
**Input:** Relation $R$ with attribute $r(a)$, Relation $S$ with attribute $s(b)$
**Output:** Resulting relation $Q$ containing matching pairs from $R$ and $S$
$H \leftarrow$ an empty hash table of size $M$                              ▷ Build Phase
for each $r_i \in R$ do
    $index \leftarrow \text{hash}(r_i(a)) \mod M$
    while $H[index]$ is not empty do
        $index \leftarrow (index + 1) \mod M$          ▷ Linear probing for next empty slot
    $H[index] \leftarrow r_i$
for each $s_j \in S$ do                                                    ▷ Probe Phase
    $index \leftarrow \text{hash}(s_j(b)) \mod M$
    while $H[index]$ is not empty and $H[index](a) \neq s_j(b)$ do
        $index \leftarrow (index + 1) \mod M$           ▷ Linear probing to handle collision
    if $H[index](a) = s_j(b)$ then
        $Q \leftarrow Q \cup (H[index] \times s_j)$
```

relation, the probe side, is used to look up or "probe" the hash table. For each key in the probe side, we check if there is a match in the hash table [91]. Typically, the smaller of the two relations is chosen as the build side because building the hash table is generally more resource-intensive and time-consuming than the probing operation [24]. A pseudocode implementation of a join algorithm using linear probing can be found in Algorithm 4

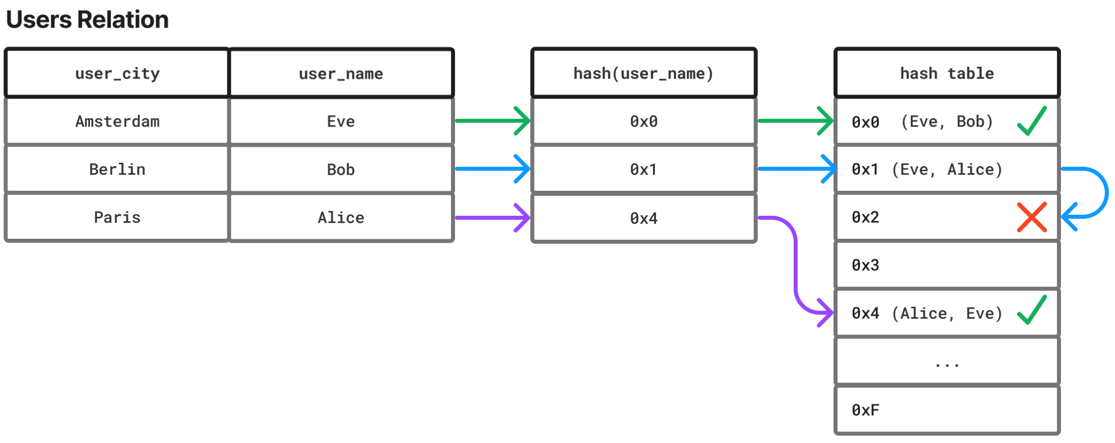
A practical step-by-step process is depicted in Figure 2.8. In this example, the hash table is first built using the `follows` relation from Figure 2.2c to join the `follows` relation with the `users` relation similar to the query of Listing 4.4 to compute the join defined as:

$$R = \texttt{Users} \bowtie_{\text{users.user\_name=follows.user\_name}} \texttt{Follows} \tag{2.10}$$

In the context of Hash Joins, the convention is to use the right-hand side of the join operation as the build side. Therefore, the columns that act as a key for the join are the `follows.user_name` on the build side and the `users.user_name` on the probe side. To build the hash table on the `Follows` relation as depicted in Figure 2.8a, the hash function is calculated for each tuple in the key column. Then, the hash table is used to store values of type (`Follows.user_name`, `Follows.follows_user_name`) using the `Follows.user_name` column as keys. As the key `Eve` occurs multiple times in the relation, when inserting the second `Eve`, we land at a hash table slot that is already occupied (blue arrow). In the



(a) Building the hash table on the `Follows` relation, using the key `user_name`



(b) Probing a hash table using the `Users` relation, using the key `user_name`

Figure 2.8: Illustration of building and probing a hash table using linear to calculate $R = \texttt{Users} \bowtie_{\text{Users.user\_id}=\text{Follows.user\_id}} \texttt{Follows}$.

example, linear probing is used for collision management. Therefore, the original index is incremented. As the following slot is free, the tuple (`Eve`, `Alice`) is inserted at index 1.

The hash table is now probed using the `Users` table using the `Users.user_name` key. The look-up starts after again calculating the hash and, therefore, the index into the hash table for each key. For the first key (green arrows), which is `Eve`, the corresponding slot with the index 0 is occupied and contains the tuple (`Eve`, `Bob`). Comparing the probe key and the key within the table results in a match, meaning the tuple (`Amsterdam`, `Eve`, `Eve`, `Bob`) can be appended to the join result.

For the second key (`Bob`, blue arrow), the hash returns the index 1 to start the look-up. The content of the slot at index 1 is the tuple (`Eve`, `Alice`), which does not match with the lookup key `Bob`. Therefore, according to linear probing, the next slot is considered. As this slot is empty, it is now clear that there is no match for this probe key. The third key in the `Users` relation is computed similarly to the first one, also resulting in a match. Therefore, the join result of this query consists of two rows:

$$\begin{aligned} R &= \texttt{Users} \bowtie_{\text{Users.user\_id=Follows.user\_id}} \texttt{Follows} \\ &= [(\texttt{Amsterdam}, \texttt{Eve}, \texttt{Eve}, \texttt{Bob}), \\ &\quad (\texttt{Paris}, \texttt{Alice}, \texttt{Alice}, \texttt{Eve})] \end{aligned}$$

In a hash join, comparisons occur when each tuple from S is used to look up matching tuples in the hash table built for R. If the hash function distributes tuples uniformly, the expected number of comparisons per probe is proportional to the number of tuples of the build side m, divided by the size of the hash table HT_{size}. The comparisons during the probe phase can be estimated as:

$$\begin{aligned} \text{Comparisons Probe S:} C_{\text{CMP, PROBE(R, S)}} &= n \cdot \frac{m}{HT_{size}} \\ m &= \text{size of build side} \\ n &= \text{size of probe side} \end{aligned}$$

Assuming $n = m$ and assuming the size of the hash table HT_{size} is chosen such that $HT_{size} \approx m \cdot 1.5$, the complexity of the hash join can be approximated as $\mathcal{O}(n)$. This is much better compared to the $\mathcal{O}(n^2)$ complexity of nested loop joins [24] and the $\mathcal{O}(n \log n)$ complexity of sort-merge joins. Therefore, hash joins generally outperform sort-based joins, but there are scenarios where sort-based joins are better. Examples are cases with non-uniform data distributions, where the data is already sorted by the join key, or when the output needs to be in a sorted order. Effective database management systems will try to use the best join implementation depending on the specific requirements of the query [24].

2.3 Graph Database Management Systems

A GDBMS is a database system that utilizes graph structures, including nodes, edges, and properties, to represent and store interconnected data. These systems are designed for high performance in graph querying featuring special operators and often also provide a graph query language designed for efficient graph manipulation and analysis. Two of these novel techniques are WCOJ algorithms, which will discussed in Section 2.3.3 and factorized representations, introduced in Section 2.3.4. Both techniques are fundamental for understanding the project on hand, as its goal is to implement these into DuckDB. In

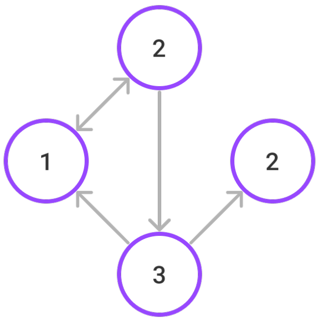


Figure 2.9: Example of a directed graph. Edges are directed: there is no relation from to (3) to (2), but there is one from (2) to (3)

graph theory, a graph is an abstract structure representing a set of objects along with the connections existing between these objects. Mathematically, a graph G is defined as an ordered pair $G = (V, E)$ [98, p.57] where:

- V is a set of vertices or nodes, and
- E is s a set of unordered pairs $\{v_1, v_2\}$ of vertices, called edges.

A directed graph, or digraph, is similar but consists of a collection of vertices linked by edges with a specific direction. Here, E is a set of `ordered` pairs.Figure 2.9 depicts an example digraph $G = (V, E)$ with $V = \{v_1, v_2, v_3, v_4\}$ and $E = \{\{v_1, v_2\}, \{v_2, v_1\}, \{v_2, v_3\}, \{v_3, v_1\}, \{v_3, v_2\}\}$. A practical example of graph application is in social networks, where each user is modeled as a node, and one user following another is represented by an edge connecting the corresponding vertices.

Common use cases for GDBMS include domains where relationship and network analysis are necessary, e.g., retail, recruitment, search, and knowledge management. In retail, GDBMS can increase sales by leveraging recommendation engines that suggest products based on user behavior and preferences. In recruitment, these systems can identify potential candidates by analyzing connections within professional networks like LinkedIn [44; 87]. Using GDBMS on social network graphs enables the analysis of both direct and indirect connections among individuals and groups to evaluate and explore each user's interactions and interests [87, pp. 106-108].

2.3.1 Property Graphs and Property Graph Queries

Property graphs are extended directed graphs that allow properties to be associated with nodes and edges [8]. A property graph consists of:

- **Nodes (Vertices)**: Represent entities in the graph. Each node can have a set of key-value pairs known as properties.
- **Edges (Relationships)**: Represent relationships between nodes which can also have properties.
- **Labels**: Nodes and edges can be labeled to categorize them. Multiple labels can be applied to a single node or edge.

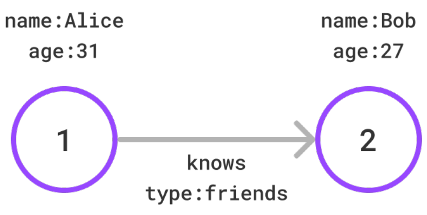


Figure 2.10: Example of a simple property graph of a social network according to [49].

For example, in a social network graph, nodes represent people, and edges represent relationships such as friendships or follows. Properties on nodes might include attributes like name, age, or location, while properties on edges might include the date the relationship was established. An example of such a property graph is depicted in Figure 2.10.

To query property graphs, one can utilize Graph Query Languages (GQLs) such as Cypher, used by Neo4j [37], or Gremlin, used by Apache TinkerPop [9]. Additionally, the SQL:2023 standard [47] includes the Property Graph Queries (SQL/PGQ) sub-language, allowing relational systems to standardize graph queries [105]. Graph query languages allow an easier definition of graph queries than default SQL.

```sql
SELECT COUNT(*)
      FROM Links AS l1
      JOIN Links AS l2 ON l1.destination = l2.source
      JOIN Links AS l3 ON l2.destination = l3.source
      WHERE l3.destination = l1.source;
```

Listing 2.1: Count of 3-hop cyclic relationships in a traditional SQL query

```
FROM GRAPH_TABLE(pg
    MATCH (a:Page)-[e1:Links]->(b:Page)
          -[e2:Links]->(c:Page)
          -[e3:Links]->(a:Page)
    COLUMNS (Count(*))
);
```

Listing 2.2: Count of 3-hop cyclic relationships using the PGQ extension in SQL:2023

A comparison between SQL and SQL/PGQ can be found in Listing 2.1 and Listing 2.2. The queries return the count of 3-hop cyclic relationships within the graph of a forum like Wikipedia, where Page a is connected to Page b, which is connected to Page c, which is again connected to a. Listing 2.1 uses traditional SQL with multiple joins to find cycles where a page links to another that links back to the original. This approach is unintuitive due to the explicit join conditions required. Conversely, Listing 2.2, using the new SQL/PGQ sub-language from SQL:2023, simplifies the expression of the graph pattern. The parentheses and arrows (`(a)-[e]->(b)`) of the SQL/PGQ syntax represent the nodes and edges, making the graph structure easier to visualize and the query easier to understand.

2.3.2 Characteristics of Graph Workloads

When executing queries on graph workloads like the one of Listing 2.2, the internal execution of GDBMS is the same as described in Section 2.1.2 on query processing: The SQL/PGQ query will be parsed and transformed into logical relational operators, optimized and then executed. One difference is that the physical implementation of these operators is heavily optimized for graph workloads [34]. According to Mhedhbi et al. [66], these workloads differ from traditional relational database workloads due to three primary reasons:

1. **Many-to-many joins** are common due to the extensive number of relationships inherent in densely connected graph data.
2. **Cyclic joins** play an important role in applications like social network recommendation systems and fraud detection, where relationships often loop back on themselves.
3. **Long acyclic joins** are utilized in path-finding scenarios, which may traverse significant depths, to analyze connections and relationships over extended paths.

Many-to-many joins as the one in Figure 2.3c are challenging for relational databases because of two reasons: the explosive nature of such joins, especially when chained, e.g., for pathfinding and cyclic joins. Even with a modest number of input tuples, these can significantly increase the volume of intermediate and final results. As an example, take the query depicted in Listing 2.2. The physical query plan DuckDB created for this query can be found in Figure 2.11. Each operator is annotated in this plan with the number of rows it produces. Here, the initial join connecting the `Links` relation to itself, which starts with 7,600,595 rows, results in an intermediate output of 222,498,869 rows, increasing by a factor of 29.2. This join retrieves all combinations of two connected edges. With its two conditions, the subsequent join closes the cycle by searching for edges that connect back to the loose ends of the previous join. Thus, it is far more selective, reducing the number of rows to 41,421,291. The final node in the tree is the count aggregation, which returns the number of rows from the preceding join, outputting a single row. GDBMSs must be optimized to handle these explosive intermediate results.

For cyclic joins, specific multi-way acwcoj algorithms can be much more efficient than the chained binary hash joins of standard relational database systems. These will be discussed in the next section.

2.3.3 Worst-case Optimal Joins

Worst-case optimal joins can be more efficient than binary join plans in finding cyclic patterns inside graphs, which is a common task in graph workloads [89]. Binary joins for cyclic queries often generate intermediate results that are larger than the maximum number of rows in the final output. This maximum size of the intermediate result is theoretically limited by the AGM bound [13]: For the triangle query $R(a,b), R(b,c), R(c,a)$, where $|R| = N$, the AGM bound is $N^{\frac{3}{2}}$. However, a binary join plan will first evaluate the open edge $R(x,y), R(y,z)$ before closing the cycle, which could generate N^2 intermediate results, which is larger than the AGM bound of the total query of $N^{\frac{3}{2}}$.

Figure 2.13 shows two examples of cyclic queries. The most simple is the triangle query $Q_{\triangle}$ depicted in Figure 2.13a. Here, we search for triangles where we have a node x which is connected to both node z and y, and node y has an edge to node z, closing the cycle. Figure 2.13b shows the more complicated Diamond-X query Q_X. In this query, we search for two connected triangles. For Q_X, there are many ways to compute the query combining binary joins with potential 3-way or 4-way joins.

Unlike binary joins that process cyclic join tables one at a time, WCOJs evaluate queries attribute-by-attribute within a single n-way join operator. The fundamental operation in WCOJs algorithms involves intersecting sets of values, which, in the context of graph processing, corresponds to intersecting adjacency lists [65].

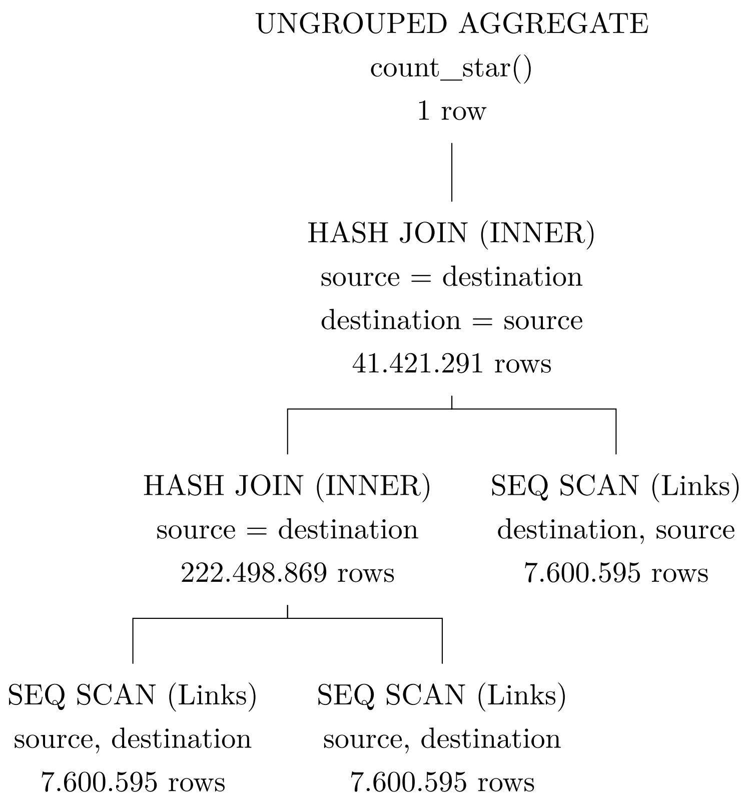


Figure 2.11: Physical Plan for querying 3-hop cyclic relationships, annotated with intermediate row counts. We can see that the intermediate has 30 times more rows than the original relations, but the final result is only one row

A classical implementation for WCOJ is the Generic Join. It was introduced by Hung et al [77] and is a WCOJ algorithm that consists of the following three steps: (1) The algorithm begins by selecting an order for the query variables. This order determines the sequence in which the attributes will be joined. This step is referred to as Join Attribute Ordering (JAO). (2) For each variable in the selected order, the algorithm intersects the sets of values corresponding to that variable from the different relations involved in the query. (3) After intersecting the sets of values for all variables, the remaining tuples are combined to form the final join result [77].

To ensure the runtime efficiency for the Generic Join and ensure it remains worst-case optimal, any implementation of the algorithm must utilize indexes that represent a trie structure on the input relations [78; 77]. An index is needed for both querying relations using a specific key (e.g., $S[y]$? in Figure 2.14 which queries for all tuples in S that fulfill y) and also to perform the set intersection. The primary contribution to the overall runtime

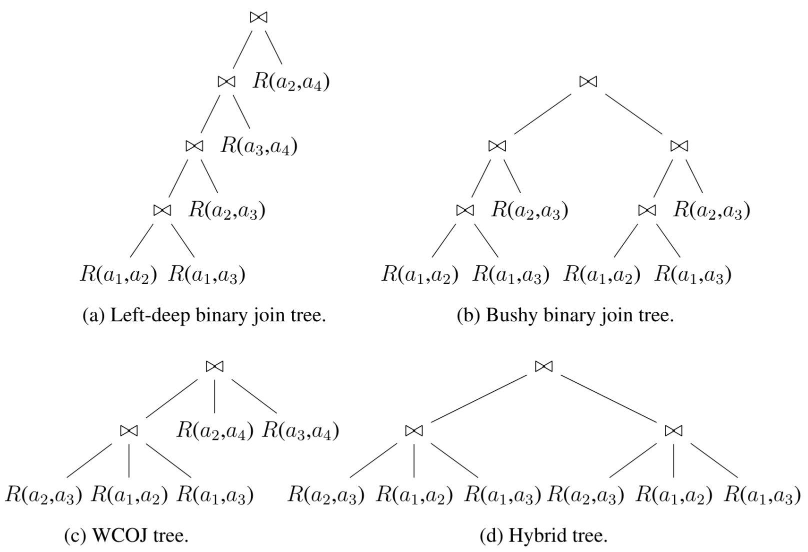

$\bowtie$
$R(a_2,a_4)$
$R(a_3,a_4)$
$R(a_2,a_3)$
$R(a_1,a_2)$ $R(a_1,a_3)$
(a) Left-deep binary join tree.
(b) Bushy binary join tree.
(c) WCOJ tree.
(d) Hybrid tree.

Figure 2.12: Example of two binary join trees (a) and (b), a WCOJ tree (c), and a hybrid join tree evaluating (d) the diamond-X depicted in Figure 2.13b [65]. Query optimizers must consider all possible combinations of hybrid query plans.

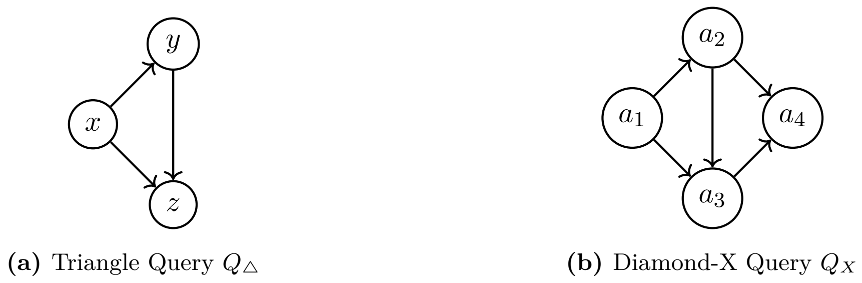

x
y
z
a_1
a_2
a_3
a_4
(a) Triangle Query $Q_\triangle$
(b) Diamond-X Query Q_X

Figure 2.13: Two examples of cyclic queries, the example was taken from [65]

of the Generic Join operation comes from the set intersection computation [3]. The most efficient way of multi-parameter set intersection is trie indexes. Traditional structures such as B+-trees or simple sorted lists are less expensive to construct but have bad performance for this task [39]. Figure 2.14 shows a pseudo-code execution for the triangle query $Q_\triangle$ depicted in Figure 2.13a. The figure lists both a binary join and a WCOJ implementation.

Unlike binary joins, where optimizing the sequence of table joins is important, WCOJ requires determining the order in which attributes are intersected. While all orderings of attributes satisfy the AGM bound restriction [77], in practice different orders lead to very different runtimes [65].

One big problem is also how to combine WCOJ with existing binary joins, as many different combinations can result in very different runtimes. For the Diamond-X Query Q_X

Algorithm 5 Binary join

```
for $(x, y)$ in $R$ do
    $s \leftarrow S[y]?$
    for $(y, z)$ in $s$ do
        $t \leftarrow T[x, z]?$
        for $(x, z)$ in $t$ do
            output$(x, y, z)$
```

Algorithm 6 Generic Join

```
for $a$ in $R.x \cap T.x$ do
    $r \leftarrow R[a]$;
    $t \leftarrow T[a]$
    for $b$ in $r.y \cap S.y$ do
        $s \leftarrow S[b]$
        for $c$ in $s.z \cap t.z$ do
            output$(a, b, c)$
```

Figure 2.14: Implementation of binary join and generic join for $Q_{\triangle}$. The expression $S[y]?$ queries S using y as a key, proceeding to the next iteration of the outer loop if no match is found. The binary join processes tuples, while the generic join processes values based on key intersections. Example taken from Wang et al. [103].

of Figure 2.13b, four of these options are visualized in Figure 2.12. It lists two options for computing the query only using binary joins. One left-deep plan, where we start by joining $R(a_1, a_2)$ and $R(a_1, a_3)$ and then joining all other relations sequentially. The second binary join plan is bushy, meaning that we have two main branches that join three relations each and then join their result with another binary join. As an alternative to these binary joins, Figure 2.12c shows a plan using only two triangle joins to compute the result. As a 4th option, Figure 2.12d sows a hybrid approach: We first compute the two triangles independently using two multiway WCOJ joins and then use a binary join to combine the two results.

Orders (O for short)

customer	day	dish
Elise	Monday	burger
Elise	Friday	burger
Steve	Friday	hotdog
Joe	Friday	hotdog

Dish (D for short)

dish	item
burger	patty
burger	onion
burger	bun
hotdog	bun
hotdog	onion
hotdog	sausage

Items (I for short)

item	price
patty	6
onion	2
bun	2
sausage	4

(a) Database with three relations

O(customer, day, dish), D(dish, item), I(item, price)

customer	day	dish	item	price
Elise	Monday	burger	patty	6
Elise	Monday	burger	onion	2
Elise	Monday	burger	bun	2
Elise	Friday	burger	patty	6
Elise	Friday	burger	onion	2
Elise	Friday	burger	bun	2
...	...	...	...	...

(b) Natural Join

Figure 2.15: Database with three relations `Orders (O)`, `Dish (D)`, and `Items (I)` (left) and the flat representation of the natural join of the three relations containing redundancy (right), e.g. the tuple $\langle Elise \rangle \times \langle Monday \rangle \times \langle burger \rangle$ is repeated three times. [33]

2.3.4 Factorized Representations

Factorized representations are a method of lossless compressing data on relationships by using algebraic factorization, allowing for more efficient data representation and processing [65, p. 5]. Factorization is based on relational algebra, especially how the Cartesian product distributes over union, which is the basis for algebraic factorization.

A good example to explain factorization is from the FDB Research Group of the University of Zurich [33]: Here, there are three relations `Orders (O)`, `Dish (D)`, and `Items (I)` which are depicted in Figure 2.15a. Applying the natural join of the three relations results in 12 rows with five columns as depicted in Figure 2.15b. As depicted, one can notice that a lot of the before normalized data is now included multiple times, e.g. the tuple `(Elise, Monday, burger)` is included three times in this flat representation of the data. Relational algebra can eliminate this redundancy by expressing the relation through the application of the *union* and *Cartesian product* operators on singleton relations, which are defined as relations containing a single column and a single row. By utilizing the distributive property of multiplication over addition, common sub-expressions can be factored out, thereby simplifying the algebraic expression [79].

$$\begin{aligned}
&\langle Elise\rangle \times \langle Monday\rangle \times \langle burger\rangle \times \langle patty\rangle \times \langle 6\rangle \vee \\
&\langle Elise\rangle \times \langle Monday\rangle \times \langle burger\rangle \times \langle onion\rangle \times \langle 2\rangle \vee \\
&\langle Elise\rangle \times \langle Monday\rangle \times \langle burger\rangle \times \langle bun\rangle \times \langle 2\rangle \\
&= \\
&\langle Elise\rangle \times \langle Monday\rangle \times \langle burger\rangle \times \\
&(\langle patty\rangle \times \langle 6\rangle \vee \langle patty\rangle \times \langle 6\rangle \vee \langle patty\rangle \times \langle 6\rangle)
\end{aligned}$$

Representing a set of tuples within the DBMS would enable us to have 9 instead of 15 singular tuples in memory. Factorization of tuples can be achieved in many different order of factors. An alternative order to the one above is depicted in Figure 2.16: The factorized representation (f-representation) presented on the right is determined by a structure on the left. This structure is defined by a partial order on the query variables, known as a variable order. It is also called an f-tree, which stands for factorization tree [15]. In general, an f-tree T can be used to describe the structure of an f-representation [65].

Initially, all potential dishes are enumerated. In the next step, for each specified dish, the lists of days on which the dish was ordered and the list of component items of the dish are considered independently. Beneath each day, an enumerating customer who placed

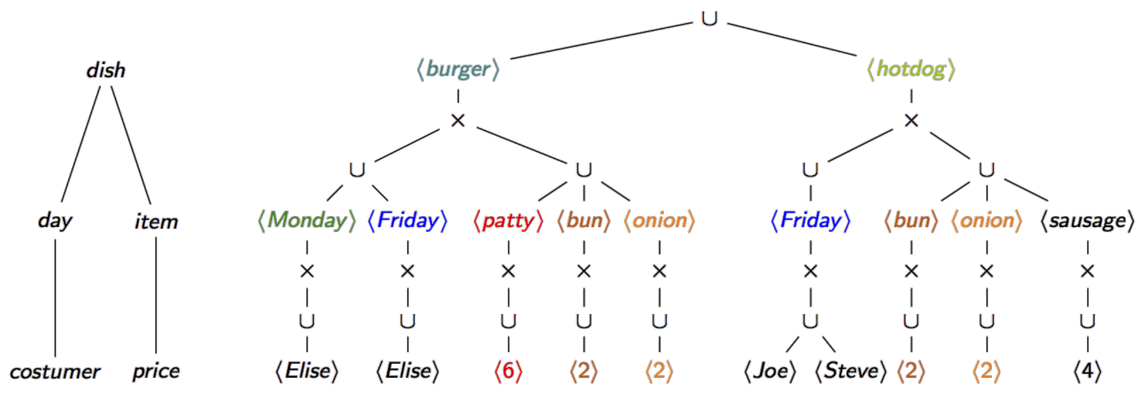

dish
day
item
costumer
price
⟨burger⟩
⟨hotdog⟩
⟨Monday⟩ ⟨Friday⟩ ⟨patty⟩ ⟨bun⟩ ⟨onion⟩
⟨Friday⟩ ⟨bun⟩ ⟨onion⟩ ⟨sausage⟩
⟨Elise⟩ ⟨Elise⟩ ⟨6⟩ ⟨2⟩ ⟨2⟩
⟨Joe⟩ ⟨Steve⟩ ⟨2⟩ ⟨2⟩ ⟨4⟩

Figure 2.16: Variable order of a factorization of the join depicted in Figure 2.15b (left) and the corresponding tree structure (right) according to [33]

.

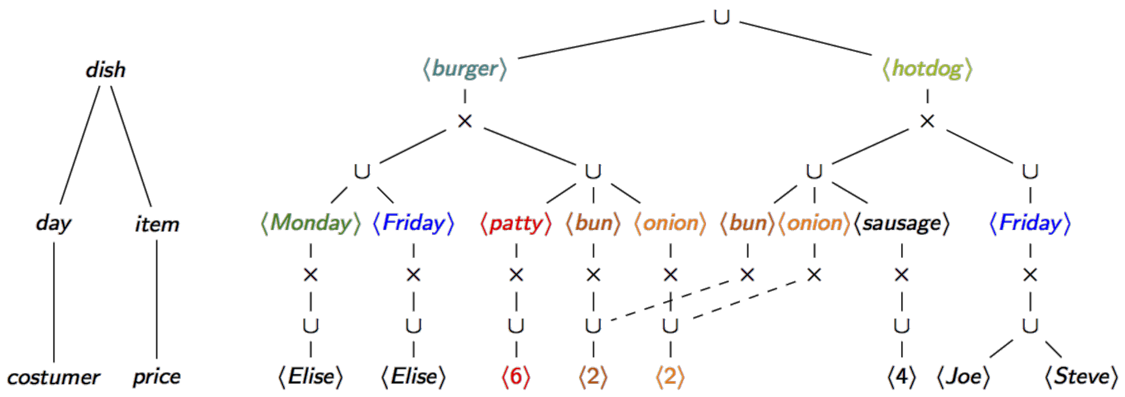

dish
day
item
costumer
price
⟨burger⟩
⟨hotdog⟩
⟨Monday⟩ ⟨Friday⟩ ⟨patty⟩ ⟨bun⟩ ⟨onion⟩ ⟨bun⟩ ⟨onion⟩ ⟨sausage⟩ ⟨Friday⟩
⟨Elise⟩ ⟨Elise⟩ ⟨6⟩ ⟨2⟩ ⟨2⟩ ⟨4⟩ ⟨Joe⟩ ⟨Steve⟩

Figure 2.17: D-representations of the f-representation depicted in Figure 2.16: As the price of the ⟨*bun*⟩ and ⟨*onion*⟩ tuple are independent of the three above, they are replaced with their definitions [33]

.

an order for the dish on that particular day is added. Similarly, the price of each item is positioned below the item itself. [33].

In f-trees, each variable is assumed to depend on all its ancestor variables, resulting in tree-structured factorizations. However, this assumption is often not true; for example, the price does not depend on the dish if the item is given. Generally, a variable may only depend on a subset of its ancestors. D-trees refine f-trees by explicitly associating each variable with a dependency set, consisting of the ancestor variables on which it or its descendants depend. This refinement can lead to more compact factorizations, as depicted in Figure 2.17 [81].

2.4 Optimizations for Read-Heavy Workloads

To increase CPU speeds, modern CPUs feature *instruction pipelines*, which break down the execution of a single instruction into multiple simpler stages. For efficient operation, the CPU tries to predict the next instructions to keep the pipeline full. If it predicts incorrectly, the pipeline must be cleared and restarted, which causes a loss in performance, particularly with longer pipelines [20].

In there are conditional branches, the CPU speculatively executes one possible path. If its prediction is incorrect, a *branch misprediction* occurs, leading to the pipeline being emptied and a loss in performance. Additionally, *instruction independence* is crucial for pipelines. For example, $a = b + c$ and $d = b \times c$ can be processed independently using two units. However, in $a = b + c$ followed by $d = a \times c$, the computation of d must wait until a is calculated [20; 108]. This is even more relevant with super-scalar CPUs, which have multiple pipelines and can execute several independent instructions simultaneously [20].

In database systems, branches dependent on data conditions, such as a filtering operator with moderate selectivity, are hard to predict, which can notably slow down query execution. To avoid branch miss predictions and data dependencies, modern DBMS feature a *vectorized* query execution engine. Here, operations do not handle single tuples, but large blocks of thousands of tuples called vectors [20]. In addition, this method decreases the number of function calls and allows CPUs to use parallel processing, loop unrolling, and the use of single instruction, multiple data (SIMD) instructions [54]. The latter allows a single instruction to be applied to multiple data points simultaneously, for example, four parallel additions.

Modern database systems need to effectively utilize not just a single CPU core but multiple. For this, systems such as DuckDB and Umbra adopt an approach known as *morsel-driven parallelism* [55]. This method involves dividing the query execution into small, manageable units of work called morsels. The key advantage of this approach is to ensure an even distribution of work and maximize throughput by implementing a task scheduling system that is both fine-grained and adaptive. In addition, this scheduling is NUMA-aware, which enhances performance by maintaining data locality and minimizing memory access delays in non-uniform memory architectures [55].

2.5 DuckDB and DuckPGQ

DuckDB, which is the RDBMS used for implementation and experimentation in this thesis, is an analytical, relational, in-process database[84]. It is freely available under the MIT License and open source[1][32]. This database management system is feature-rich, offering a “friendly” SQL dialect [69] and integrations with different file formats like comma-separated values (CSV), Parquet, and JavaScript Object Notation (JSON).

As an analytical system, DuckDB implements many of the optimizations for read-heavy workloads of modern DBMS [14]:

- **Columnar storage:** DuckDB uses skippable columnar storage using MinMax statistics and a variety of lightweight compression schemes, see.
- **Indexing:** Supports primary and foreign keys backed by the ART index [56], optimizing data retrieval.
- **Push-based Vectorized Query Execution:** DuckDB has a vectorized execution engine that operates directly on compressed data and a push-based operator execution driven by a morsel-driven pipeline scheduler.
- **Robust Out-of-Core Operations:** The operations for join, aggregation, and sorting in DuckDB are designed to run efficiently out-of-core, with a gradual decline in performance as memory becomes limited [52].
- **Join Optimizations:** Uses HyperLogLog for enhanced statistics and cardinality estimations.
- **Morsel-driven Parallelism:** DuckDB supports NUMA-aware multi-threaded execution.

As a vectorized query execution engine, DuckDB’s operators work primarily on groups of vectors that contain 2048 tuples. These groups of vectors are called `DataChunk` and contain one vector for each column they represent. More complicated data types like lists or structs can be represented as nested vectors. For example, to represent a list, DuckDB uses one vector of size 2048 which contains indexes to another, variable-sized vector, which contains the elements of the lists. For filter-like operations on a Vector, DuckDB uses `SelectionVector`s which contain the indices of the “active” tuples in the Vector.

[1]`https://github.com/duckdb/duckdb`

DuckDB can be extended with extensions, which can add new data types, functions, file formats, and SQL syntax enhancements. One of these extensions is DuckPGQ[2], built to support graph workloads and the SQL/PGQ standard [105]. It is freely available under the MIT-License and open source. DuckPGQ builds on the optimizations of DuckDB but also incorporates additional optimizations like fast on the fly compressed sparse row (CSR) generation and custom operators for path finding [106; 86]. Because of these optimizations, DuckPGQ can outperform the popular GDBMS Neo4j in pattern-matching and path-finding queries [106]. To further optimize DuckPGQ and DuckDB for graph workloads, this project aims to experimentally implement factorization and WCOJ in DuckDB and DuckPGQ.

[2]`https://github.com/cwida/duckpgq-extension`

3

Literature Review

The following chapter will review current literature concerning the adoption of WCOJ and factorized representations in DBMSs. In Section 3.1, we will list the systems that currently implement WCOJ algorithms. In Section 3.2, we will do the same for factorized representations. We will then summarize our findings and list the elements from the literature that can be applied to our approach. This can be found in Section 3.3. Background information on factorization and WCOJ is provided in Table 3.2 and Section 2.3.3 respectively.

3.1 Adoption of Worst-case Optimal Joins

This section will list which systems feature Worst-case Optimal Joins and which challenges they faced when implementing them. Note that we will not include the Dovetail WCOJ of RelationalAI [18], as we did not find any academic source describing the specific algorithm in detail.

3.1.1 GraphflowDB, later Kùzu

GraphflowDB is an in-memory, non-transactional graph database used as a research prototype. Kùzu [34] is the follow-up system developed by the Data Systems Group at the University of Waterloo. Unlike GraphflowDB, Kùzu is designed to be a fully functional, user-facing, and highly scalable production-ready GDBMS. Kùzu is a disk-based system that scales beyond memory and offers more robust and scalable join capabilities than GraphflowDB [34].

GraphflowDB [65] uses the Generic Join algorithm [77] as their WCOJ algorithm [65]. For binary joins, Graphflow users both `Index Nested Loop Joins` or `Hash Joins`. The `Index Multiway Join` implements the Generic Join algorithm discussed in Section 2.3.3. For list intersection, GrapflowDB assumes that all binary relations used for the join have an index that allows fast retrieving of tuples in this relation that satisfy a certain condition [65].

The system allows the optimization of hybrid query plans containing both WCOJ and binary joins. Each attribute ordering for the WCOJ is considered a separate plan. These plans are then optimized through a metric called intersection cost. It estimates the intersection workload in intersecting adjacency lists during query execution and uses this assessment to rank the performance of different query plans based on this heuristic.

The system also incorporates the cost of binary joins when ranking hybrid query plans. This holistic approach considers the intersection cost and the costs associated with binary joins. The optimizer can adapt to the characteristics of the input graph. It considers variations in the sizes of forward and backward neighborhoods and the expected output sizes of subqueries. Therefore, Grapflow will provide different query plans for the same query on different data [65].

However, this cost-based method depends on statistics from subsets of the base tables [65], which can result in cardinality estimation errors during query planning, leading to suboptimal plan choices. These inaccuracies tend to increase with more downstream operations, particularly filtering, which can significantly alter the data properties of the filtered subsample. Therefore, planning operators above the filter can be very difficult.

GraphflowDB handles parallel query execution through work-stealing-based, morsel-driven parallelization [55]. Each worker or thread in the system is provided with the physical plan. Workers operate independently by fetching tasks from a common queue. Threads can execute extensions in the `Index Nested Loop Joins` and `Index Multiway Join` operators independently, without needing to coordinate with one another.

3.1.2 Umbra

Umbra [76] is disk-based RDBMS using code generation for query execution. The system features an implementation of WCOJ for general-purpose RDBMS, which might not have only binary, easy-to-index relations like typically GDBMS. This is done with a hash-based worst-case optimal join algorithm, utilizing a hash trie structure constructed during query execution [39].

This Hash Trie Join Algorithm is divided into two distinct stages: the build and probe phases. During the build phase, input relations are materialized, and hash tries are constructed. Following this, in the probe phase, the worst-case optimal hash trie join algorithm uses these index structures to generate the join results [39].

Many systems like GraphflowDB or EmptyHeaded assume that input data is sorted or indexed, which can be costly to maintain in practical scenarios. For example, GraphflowDB only computes joins on binary relations and therefore only requires indexing these relations twice [65]. For more complex n-ary relationships, we must maintain up to $n!$ different indices to support all possible attribute orderings. As this is very costly, it is hard to justify, especially if the system should also support updates of the data [39]. For example, Freitag et al. measured that the EmptyHeaded system needs up to two orders of magnitude more time to precompute the index than the actual join processing [39].

Umbra introduces a multiway join operator that dynamically computes the trie indexes needed for its WCOJ algorithms during query execution. These indexes are structured as nested hash tables, where each level represents a join key attribute, and the leaf nodes are organized using a linked list [39].

Freitag et al. note that despite the advancements in WCOJ algorithms, these often do not match the performance of traditional binary joins on queries without growing joins [38]. For example, the highly optimized LevelHeaded system [2] is outperformed by HyPer [50] by a factor of up to two on certain TPC-H Business Intelligence (BI) queries, even when precomputation costs are excluded [39]. Furthermore, benchmarks show that Umbra surpasses a commercial system utilizing worst-case optimal joins by up to four orders of magnitude on TPCH and JOB benchmarks using binary joins [39].

Therefore, Umbra applies a heuristic method that enhances optimized binary join plans by substituting cascades of potentially growing joins with worst-case optimal joins. These plans prevent the growth of intermediate results, improving upon the original binary plans. Growing joins are identified using the same cardinality estimates applied in regular join order optimization [39].

3.1.3 BiGJoin in Timely Dataflow

Timely Dataflow is a distributed system designed for executing data-parallel, cyclic dataflow programs [71]. Ammar et al. [7] developed a variant of the GenericJoin algorithm, named BiGJoin, which they implemented on the Timely Dataflow platform. With BiGJoin, the authors developed a distributed operator that ensures cumulative worst-case optimality, effective workload balance, and low memory usage per worker across real-world datasets and queries [7].

As BiGJoin implements the WCOJ algorithm GenericJoin for a distributed system, there are challenges regarding the cooperation of different nodes. A key challenge in this setting is managing memory requirements and computational load on individual machines while minimizing communication. This gets more complicated as the number of workers increases or if the data is skewed. Ammar et al. tackle this with a batching technique for memory control and an intersection planning method that prioritizes shuffling smaller adjacency lists to reduce communication [7].

3.1.4 LogicBlox

The LogicBlox system is a commercial [101] tool that uses a logic programming language and a deductive database. The language is inspired by Datalog to handle data in a declarative and incremental way [11]. Datalog is a declarative logic programming language and a syntactic subset of Prolog. It allows for deriving additional facts by applying rules to the existing facts stored within its database [21]. Listing 3.1 shows an example that defines the ancestor relationship and queries for all ancestors of 'john' using Datalog.

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

% Example query: Find all ancestors of 'john'
ancestor(X, john).
```

Listing 3.1: Example of a Datalog query

The LogicBlox System uses a WCOJ algorithm called Leapfrog TrieJoin [102]. It is optimized for evaluating joins across general relations. Like the GenericJoin, this algorithm relies on multiway intersections to perform the joins [102]. The process starts by placing an iterator at the beginning of each predicate. Then, it follows the following rules: (1) If either iterator reaches the end, the process stops. (2) If both iterators are at the same key, that key is emitted, and one of the iterators is advanced. (3) If the iterators are on different keys, the one at the smaller key is moved forward to match the position of the other iterator. This process repeats until the join is complete [101]. To perform this sort of intersection, all predicates need to be sorted. Therefore, the algorithm assumes that the input data is indexed according to a global attribute ordering and selects a join attribute order that aligns with this global attribute sequence [65]. The LogicBlox query optimizer uses small samples of predicates to compare variable orderings for Leapfrog TrieJoin evaluation, also used for automatic index creation [11].

The LogicBlox system does not use binary join operators, focusing exclusively on generating and executing plans featuring the Leapfrog TrieJoin [65].

Incremental maintenance is a key feature of the LogicBlox applications [11]. Incremental maintenance of derived predicates is handled by extending the LFTJ algorithm using *trace* *maintenance*. In this approach, queries are represented as a computational graph. For example, the equation $r = (a + b) * c$ would result in a graph where the nodes represent the variables a, b, and c, and the edges represent the operations between them. In this computational graph, changes to the inputs automatically propagate through the graph, updating the result. If c is updated now, only a subpart of the graph has to be recomputed - the result of the addition $a + b$ stays the same. To use this approach for the Leapfrog Triejoin, LogicBlox maintains traces on an operation of the individual iterators of the join algorithm [101].

3.1.5 EmptyHeaded

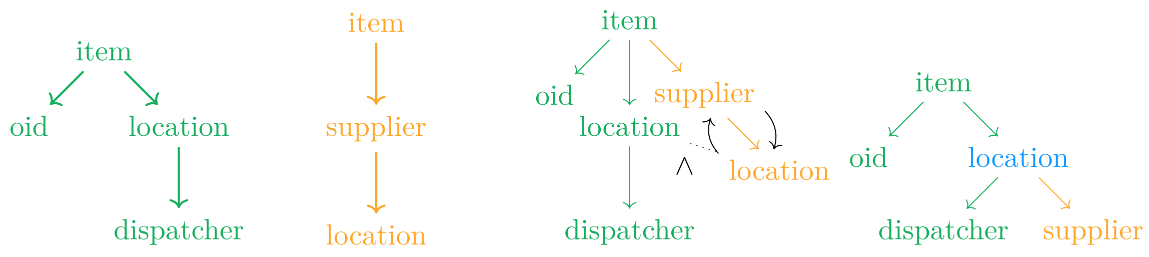
EmptyHeaded [3] is a relation system designed to evaluate graph workloads. The system focuses on using SIMD parallelism in their WCOJ algorithms. Like GraphflowDB, it uses the Generic Join algorithm but implements it in a SIMD friendly way. For instance, EmptyHeaded improved the set intersection for two sorted unsigned integer lists by optimizing for SIMD and reducing branch mispredictions. Building on the LogicBlox system's merge-based approach, it compares multiple elements from two input arrays simultaneously (step (2) of LogicBlox), enhancing efficiency by reducing unpredictable conditional branches as they can now step the iterators more than only once [48].

3.2 Adoption of Factorized Representations

While certain industrial-strength systems have already adopted WCOJ algorithms, the adoption of factorization is less widespread. To our knowledge, only two systems broadly support factorized representations: The FDB system [15] developed at the University of Oxford and GraphflowDB [65], which became Kùzu [34], developed by researches at the University of Waterloo. Beyond that, there are approaches to integrate factorization in relational systems by using special (temporary) structures built during query processing. These include the 3D-Hash Join paper [35] and Umbra with the Robust Join Processing paper [19].

3.2.1 FDB System

The FDB system [15] was the first to introduce factorized query processing. It is an in-memory engine designed for handling select-project-join queries.



(a) T_1: Relation R_1 **(b)** T_2: Relation R_2 **(c)** T_3: $R_1 \bowtie_{\texttt{item}} R_2$ **(d)** T_4: $R_1 \bowtie_{\texttt{item,location}} R_2$

Figure 3.1: Challenges when joining two factorized relations according to [15]: We want to compute $R_1 \bowtie_{\texttt{item, location}} R_2$, because of the f-trees, $R_1 \bowtie_{\texttt{item}} R_2$ is simple to compute by intersecting `item`. For the condition `location`, it could be more efficient to rearrange T_2 for a faster join computation.

FDB uses f-representations and works on trie structures. It indexes all input relations as tries, and its execution plans consist of a linear sequence of operators that transform and produce tries, starting from the ones that represent the input data [15].

Bakibayev et al. aim to find f-trees that provide the most succinct representations of query (intermediate) results for databases, whether the input is flat or factorized. The query and the schema determine these f-trees. The query optimizer focuses on two main metrics: reducing the computing cost of the factorized query result and minimizing the size of the output [15].

The search for an effective query and factorization plan (f-plan) additionally requires considering operators beyond the standard selection, projection, and product. These operators are concerned about reordering an f-tree to speed up the following query operators. The FDB system has two of these operators. The swap operator exchanges a child with its parent in an f-tree, and the push-up operator moves up an entire subtree within the f-tree. This example can be seen in Figure 3.1. Bakibayev et al. illustrate this with the following example: We want to compute the join $R_1 \bowtie_{\texttt{item, location}} R_2$. The f-tree T_1 of R_1 is depicted in Figure 3.1a, the one of R_2 is depicted in Figure 3.1b. For these f-trees, joining on the `item` column is very efficient, as both T_1 and T_2 have this column as their first variable. Therefore, the `item` column is on the first hierarchical level of the try, which makes it easy to intersect. The resulting f-tree of this join $R_1 \bowtie_{\texttt{item, location}} R_2$ is T_3 and depicted in Figure 3.1a. If we want to check for the second join condition, `location` on T_3, we would need to iterate each of the suppliers and then access all the locations in the orange subpart of the tree. The join could be much easier to compute if the location and supplier nodes of T_2 were swapped, as indicated by the black arrows. This would allow an

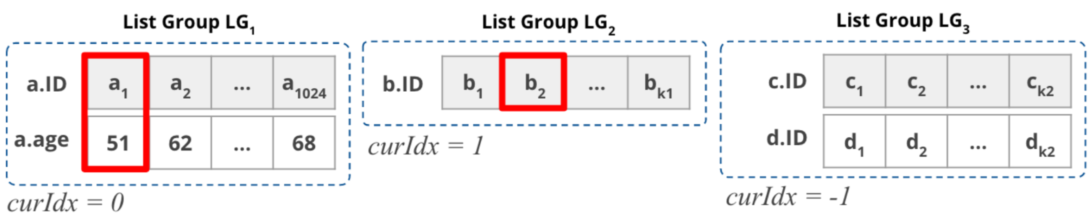


Figure 3.2: Factorized Representation in Kuzu: The first two *list groups* are flattened to single tuples, while the last one represents k_2 many tuples [45].

easy computation of the full join, resulting in T_4 depicted in Figure 3.1b. However, also this swapping is computationally expensive. The query optimizer must be able to evaluate both options to get an optimal plan[1].

3.2.2 GraphflowDB, later Kùzu

GraphflowDB uses multiple groups of vectors, referred to as *list groups*, to represent intermediate data. Each *list group* possesses a field named *curIdx* and can exist in one of two states:

- **Flat**: The *list group* is considered flattened if $curIdx \geq 0$. This state represents a single tuple consisting of the *curIdx*-th values from the blocks.
- **Unflat list of tuples**: The *list group* is represented as an unflat list of tuples if $curIdx = -1$. This state allows the *list group* to represent as many tuples as the number of elements within its blocks.

Additionally, in its unflat state, the vector lengths of a tuple can exceed the default fixed-length vector size, such as 1024, extending up to the lengths of adjacency lists in the database. This approach eliminates the need to materialize adjacency lists into blocks. Figure 3.2 depicts an intermediate chunk composed of three *list groups* [45]. The first two groups are in a flattened state, whereas the last one is in an unflat state. In this state, the intermediate chunk embodies k^2 intermediate tuples, represented as $(a_1, 51) \times (b_2) \times ((c_1, d_1) \cup \ldots \cup (c_2, d_2))$. In the factorization theory, this representation is known as the f-representation [34; 79].

Contrary to using a single Join operator, such as Neo4j's Expand, which utilizes the adjacency list indices to implement the index nested loop join algorithm, two joins are employed.

[1]Empirical evaluation shows

Factorized Vectors are obtained by joins over 1-n or n-n edges e by a `ListExtend` operator. Consider nodes a connected through 1-n or n-m edges e to nodes b. The input list group LG_a, containing vectors of values corresponding to a, may be presented in a flat or unflat format. If LG_a is unflat, it is first flattened by setting the `curIdx` field of the list group to 0. Each a value, denoted as a_l, is processed in a loop, where it is extended to a set of b and e values using its adjacency list Adj_{al}. These b and e values are then placed into a new list group, LG_b, which segregates a list of b and e values for an individual a value. The block lengths in LG_b, for both b and e values, have the same length as Adj_{al}, differentiating from the fixed block sizes seen in traditional block-based processors [45]

The second join operator, `ColumnExtend`, is used for 1-1 or n-1 joins. Instead of adding a new list group to obtain a factorized representation, the input list group is extended with new column vectors. This can be done as the maximum cardinality of the result will stay the same. [45]

Furthermore, GraphflowDB also features implementations for d-representation using nested hash tables. Therefore, GraphflowDB adds a blocking operator called `DGroup-by` that builds progressively these nested hash tables by appending multiple `DGroup-by(s)` to a WCOJ subplan [65]. The second key component is directed, acyclic graph-style plans, where `Index Nested Loop Joins` or `Index Multiway Join` operators share d-representations produced by `DGroup-by` operations. These operators can skip parts of the plan if a sub-relation for a specific key has already been computed, moving directly to the child of the last `DGroup-by` used. If the sub-relation has not been calculated, the computation continues with the next child of the `Index Nested Loop Joins` or `Index Multiway Join` operator.

In existing block-based processors, binary expressions typically process two blocks of values. However, the list-based processing of GraphflowDB demands a more capable filter operator that can handle three potential value combinations: two flat values, two lists, or a combination of one list and one flat value. However, it can benefit from the factorized format by not needing to process as many tuples [65].

GraphflowDB computes `GROUP BY` and aggregations based on their factorized representations. This is highly efficient, as the computation of `count(*)` is possible by multiplying the sizes of list groups, thereby determining the tuple count represented by each received intermediate chunk.

Real-world graph data with *n-n* relationships often show power-law degree distributions [59], leading to many short adjacency lists. For example, FLICKR and WIKI graphs have average degrees of 14 and 41, respectively, while the commonly used Twitter dataset has a degree of 35 [53; 45]. When processing queries with multiple joins, reading these lists from memory involves iteratively reading short lists and performing random accesses to get the node information. For the storage layer, compression techniques that decompress data blocks of a few KBs to read a single vertex property or short adjacency list can be expensive [45]. Furthermore, this skew in the factor length could lead to workload distribution issues, which can be problematic for parallel computation.

3.2.3 Umbra

Birler et al. incorporate factorized representations implicitly into Umbra in the diamond-hardened join framework [19]. In traditional in-memory binary hash joins, a hash table is built on one relation and probed with another to produce all matching pairs. This process, while common, can be resource-intensive when dealing with n:m joins that often result in duplicate values. Most existing systems miss out on these potential optimizations because they immediately materialize the join result, flattening the factorization and increasing the computational load. The authors propose to exploit this natural factorization by splitting up joins in a `Lookup` and `Expand` phase [19]. This is similar to *deferred unnesing* of the 3D hash join proposed by Flachs at al. [35].

An SQL query optimizer must handle inner joins, outer joins, semi-joins, and anti-joins. Unlike inner joins, which can be freely reordered, outer joins are more complicated because they produce null values and are not commutative or associative. This makes reordering difficult. Birler et al. suggest using the Lookup and Expand decomposition for outer joins. By making Lookup the operator that produces nulls, the optimizer can push Expand up the join tree more easily. This approach increases flexibility in query planning, allowing the optimizer to explore more ordering options for Expands, even though full reorderability is not always possible [19].

Furthermore, Birler et al. propose to extend the `Lookup` and `Expand` principle to handle cyclic queries worst-case optimally. The authors introduce the `Expand3` operator. This operator can expand two iterators conditional. Combining this operator with Lookup and Expand is sufficient to achieve worst-case optimality in almost all queries [19]. This ternary operator is based on Algorithm 2 from Ngo et al. [77], using intersections implemented through hashing.

With the `Expand3` operator, Umbra can compute ternary cycles worst-case optimal as shown by the following example: Consider the cyclic query $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$. This query's Lookup and Expand plan could be represented as $e_T(e_S(R \to S) \to T)$. Therefore, the expansion happens before the second lookup, as R and S share predicates with T. If

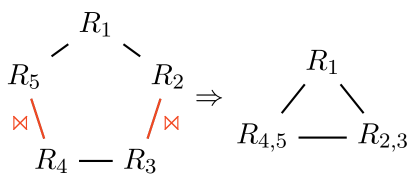


Figure 3.3: Transforming a pentagon query into a triangle query allows computing it with two standard binary joins (red) and two `Lookup`s and one `Expand3`, allowing worst-case optimal computation [19].

the inner expansion $(R \to S) \to T$ is avoided, the intermediate result retains R with two iterators, one for S and one for T. The `Expand3` operator then takes these two iterators and computes their intersection on the predicate $S.c = T.c$. The resulting plan is represented as $e3_{S,T}((R \to S) \to T)$ [19]. For this plan to be worst-case optimal, the intersection must be computed in time proportional to the length of the shortest iterator [77]. Therefore, the `Expand3` operator selects the iterator with the smallest number of tuples and performs hash-lookups into the tuples referenced by the other iterator [19].

The authors argue that only having the `Expand3` operator is enough to compute most classes of cyclic joins worst-case optionally. This is done by decomposing n-ary cycles into binary joins followed by a `Expand3` operation [19]. For a pentagon query, this is shown in Figure 3.3, where the binary joins used to transform the query in a triangle query are depicted in red. The triangle can then be computed using `Expand3`.

Birler et al. argue that it is not necessary to implement aggregation functions that directly operate on unexpanded intermediates. Instead, they suggest that join and aggregate queries, which produce intermediate results with duplicate values, can benefit from eager aggregation [19]. Eager aggregation involves calculating partial aggregates before some joins are executed, thereby reducing the cost of the subsequent joins. It is achieved using transformation rules that push the group-by operation down the join tree [107]. For instance, consider the query $\Gamma_{b;sum(c)}\,(R(a,b) \bowtie S(b,c))$. This query can be optimized by performing eager aggregation on S first:

$$Q = \Gamma_{b;sum(c)}\,(R(a,b) \bowtie S(b,c)) \tag{3.1}$$

$$= \Gamma_{b;sum(c')}\left(R \bowtie \Gamma_{b;sum(c):c'}(S)\right) \tag{3.2}$$

The `Lookup` operator finds the first match in the hash table for each probing tuple. The subsequent Expand operator then iterates over any additional matches. This separation allows the query to remain factorized until the Expand operator is executed, thereby avoiding unnecessary computations. The Umbra optimizer aims to order operators based on their impact on data size: shrinking operators like Lookup are pushed down the execution plan as they reduce the amount of data early on, while growing operators like Expand are pushed up. However, errors in the optimizer's cost estimates can still lead to poor plan choices, sometimes slowing performance by up to 21 times [19].

3.2.4 3D Hash Join

The 3D Hash Join Paper by Flachs et al. proposes a new hash table layout for hash joins that features hierarchical chains for hash collision handling [35].

Non-unique join attributes in the build phase typically result in long collision chains in traditional hash tables that use chaining for duplicate resolution, leading to increased memory accesses and expensive join predicate evaluations. To address this, a 3D hash table organizes each bucket hierarchically. Instead of a linked list for each bucket, Flachs et al. propose main collision chains and sub-chains. Each distinct key is associated with a main node that stores both the key and its initial value, while subsequent values for the same key are stored in sub-nodes linked to the main node. This hierarchical structuring reduces the complexity and number of memory accesses during the probe phase by separating the storage of keys and values between main nodes and sub-nodes, respectively. During the probe phase, if the main node corresponding to a key is found, the sub-chain linked to this node can be traversed without needing further comparisons [35].

The sub-chains of the hash table are free from collisions, enabling an optimization the authors call *deferred unnesting*. This method emits pointers to the sub-chain when probing a key rather than producing a flattened result. Therefore, the authors split the default join into a `join` and a `Unnest` operator. This optimization is used when the predicate of a secondary join solely depends on attributes from the probe side, enabling unnesting further up in the operator tree. An example of this case is the two consecutive joins $\left(R \bowtie^{3D}_{R.a=S.a} S\right) \bowtie_{R.b=T.b} T$, where the second join only depends on the probe side key $R.b$. Deferred unnesting improves the performance as the second join processes significantly fewer tuples and even may filter out nested tuples [35].

While the 3D Hash Paper does not use terms like factorized representations, the emission of pointers to collision-free hash table chains technically qualifies as d-representations. However, the proposed delayed unnesting only allows for operations on the first hierarchical layer of the d-representation, not on the definitions themselves.

Table 3.1: Comparison of WCOJ implementations across different systems

Category	**Umbra** [76]	**Kùzu** [34]	**BiGJoin** [7]	**LogicBlox** [11]
Algorithm	Hash Trie Join & `Expand3`	Generic Join	Generic Join	Leapfrog TrieJoin
Data Structures for WCOJ	Hash Trie for Hash Trie Join, Hash Table for `Expand3`	Graphflow: static index on binary relations, Kùzu: Runtime index using hash tables	?	Global attribute order assumed, JAO must be an element of this order.
Data Structure Creation	Runtime	Ahead of Runtime	Ahead of Runtime	Ahead of Runtime
Join Attribute Ordering (JAO)	Heuristic using cardinality estimates to detect growing intermediates	(adaptive) dynamic programming cost-based optimizer	Arbitrary	Sampling-based
Hybrid Query Plans	Same as for JAO above	Cost-based, dependent on the data and the query	Not supported	Not supported
Parallelization	Morsel-driven Parallism	Morsel-driven Parallism	Distributed over n-workers	Operators are parallized
Challenges Noted by the Authors	Integrate WCOJ into RDBMS without less BI performance, optimizing hybrid plans, party solved by `Expand3`.	Optimizing hybrid plans	Workload splitting with skewed data	Extended Leapfrog TrieJoin algorithm for Incremental View Maintainance

Table 3.2: Comparison of factorized representation implementations across different systems

Category	**Umbra** [76]	**Kùzu** [34]	**FDB** [15]	**3D Hash Join** [35]
f-Representation Support	Not supported	Implemented using variable sized vectors	Implemented using tries, heuristic f-tree optimization	Not supported
d-Representation Support	Supported as iterators emitted from the `Lookup` operator	Implemented using nested hash tables materialized by `DGroup-by` operator	Not supported	Supported as pointers to sub-chain in the 3D Hash Table
Supported Factorized Algorithms	Sequential and cyclic Joins.	Selections, Projections, (Multiway)-Joins, Filters, Aggregates,	Selections, Projections, Joins, Restructurings of the f-tree like Swaps and Reorder	Sequential Joins on attributes not contained in the definitions.
Challenges Noted by the Authors	Query optimization is difficult because of errors in the cardinality estimation	Skew in the data leading to big factors, Cost based optimization can lead to suboptimal plans	Finding good f-trees, Reordering f-trees during query execution	Creating hash table chains that can be used for d-representations by mitigating hash collisions

3.3 Challenges in Adopting WCOJ Algorithms and Factorized Representations

In this section, we summarize the different approaches to integrating the new techniques of Factorization and WCOJs. The list of systems reviewed contains both very specialized systems like Kùzu [34], but also general-purpose systems like Umbra [76]. We aim to answer the question of how these techniques were implemented in specialized systems to find out how we can implement them in more general-purpose systems without overhead. The final goal is to find implementation approaches that we can use for our interaction of WCOJs and Factorization in DuckDB

In our literature review, we analyzed five systems that implement WCOJ joins and four systems that support factorized representations. A summary of these findings is provided in Table 3.1 for the WCOJ implementations and in Table 3.2 for the factorized implementations. We found out that both techniques are complex to integrate into modern analytical DBMSs, especially if these feature optimizations like pipelined and vector-based query execution or morsel-driven parallelism [65].

Adopting the WCOJ, algorithms are complex for general purpose DBMS as they require infrastructure for efficient computation of list intersections along columns. GDBMS like GraphflowDB and EmptyHeaded feature relations that only have two columns (`source` and `destionation`), so it is easy for these systems to maintain an index on these columns. For more general-purpose systems, relations can have many columns; therefore, maintaining an index is not feasible. Thus, such systems must rely on runtime-created data structures like the implementation of Umbra, which degrades the performance of WCOJ algorithms for these systems.

The BiGJoin implementation in Timely Dataflow showed that it is hard to parallelize their WCOJ implementation on multiple machines, especially if there is skew in the data, like in graph workloads [89]. This is also important for non-distributed systems which achieve parallelism through multithreading. If there is skew in the data, some list intersections might be less selective or feature more data than expected and, therefore, might be more expensive to calculate, leading to one thread having more work than the others.

In addition, general-purpose analytical queries must also support workloads that do not feature explosive joins, like the usual BI workload. In these cases, WCOJ are inferior to binary joins [39], so the query planner must carefully select between WCOJ and binary joins. Both Umbra and GraphflowDB approached hybrid join plan optimization using heuristic query optimizers.

We argue that the main challenges for adopting WCOJ algorithms in general-purpose systems are (1) creating and maintaining data structures for fast list intersection under changing data and (2) optimizing query plans to combine binary- and worst-case optimal joins effectively[2]. Achieving good parallelism with skewed data is not a problem specific to adopting WCOJ.

We identified two main approaches to integrating factorization. Special-purpose systems like Kùzu and FDB feature custom data structures to maintain f-representations of tuples. In the case of Kùzu, these are sizeable vectors, while FDB uses tries. Systems that also aim to perform well on default relational workloads tend to adopt d-representations by repurposing existing temporary structures created during query execution. Umbra and the 3D Hash Join use pointers to hash table chains as their d-representation. These temporary structures require (little) extra effort to support factorization, such as the hierarchical chains of the 3D-Hash Join.

The FDB system shows the benefits and challenges of finding good f-trees. Not only is it difficult to find the f-tree that gives the best compression rate, but the query that needs to be computed must also be considered when choosing an f-tree. For example, if T_2 of the example in Figure 3.1 would already have the structure `item` $\rightarrow$ `location` $\rightarrow$ `supplier` the costly swap in from T_3 to T_4 would be obsolete. Systems that integrate d-representations via temporary structures do not feature such operations.

Even without f-tree reorderings, planning query execution on factorized intermediates is challenging, particularly for general-purpose systems receiving many default BI queries without duplicates, which gain little from factorization. Consequently, the query optimizer must determine *if* and *how* to utilize factorized representations.

We find two main challenges for integrating factorized representations in general-purpose systems: (1) maintaining data structures representing factorized representations and (2) query optimization with factorized representations. Notably, these are similar challenges as in the adoption of WCOJ algorithms.

While specialized systems like Kùzu already profit from factorized representations and WCOJ [65; 34], most general-purpose systems still lack these new technologies. This is because the specialized systems can assume that because of the workload they are designed for, the new technologies will be beneficial in most cases. Therefore, the respective implementations can also be quite resource-heavy, like maintaining a constant index on the relations or processing factorization-tailored data structures like ties or lists.

[2]In our evaluation, for a simple triangle query, both Kùzu and Umbra did not choose the optimal plan. See Section 4.8.2 for more information

However, recent research shows that factorization and WCOJ do not have to be implemented explicitly as in the FDB but can be integrated more implicitly. By reusing existing structures not originally planned out to be used as factors like the chains of hash tables, general-purpose systems can adopt the new technologies. First and foremost, the team behind Umbra shows how such a more lightweight [39] and implicit implementation [19] could look, paving the way for widespread adoption.

We aim to integrate factorized representations and WCOJ into DuckDB, a general-purpose OLAP system designed for diverse workloads. Our approach will focus on lightweight implementation strategies, like the 3D Hash Join technique [35] and the Diamond Hardened Join [19].

4

Factorization using Linear-Chained Hash Tables

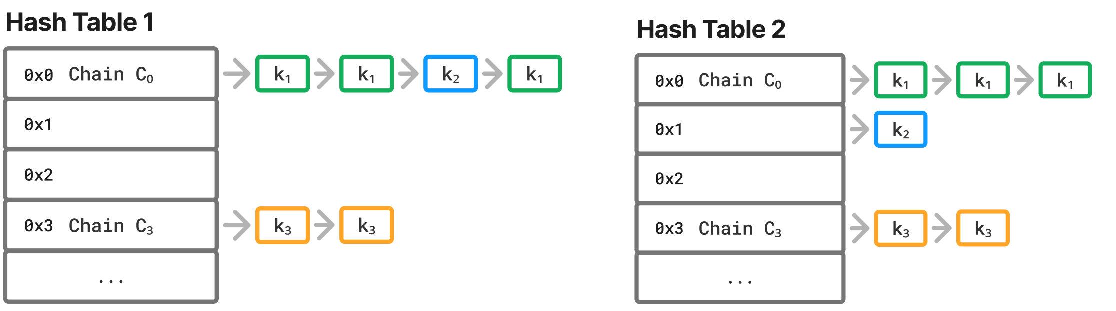
This chapter will discuss introducing WCOJ algorithms and factorized representations in DuckDB. Therefore, we present enhancements to the Hash Join Operator to enable the emission of factorized tuples. We aim to leverage hash table chains to represent factors, which will be discussed in Section 4.1. Therefore, the chapter will also introduce techniques to create collision-free chains within hash tables, as homogeneity in keys within a chain is necessary to utilize them as factors. This approach should significantly reduce the number of comparisons required during the probe phase of many-to-many joins, which will be discussed inSection 4.2. Then, we will show how we can use these factors to efficiently compute aggregates (Section 4.8.1). In addition, we propose a method to compute cyclic joins on factorized representations in WCOJ-like manner (Section 2.3.3).

The necessary background on factorized representations and WCOJ algorithms can be found in Section 2.3.4 and Section 2.3.3, respectively.

4.1 Hash Table Chains as Factors

The key idea of the project on hand is to use the hash table chains as factors for the factorized representations discussed above. Instead of flattening the list when emitting tuples from the join operator, this approach returns pointers to these lists, using the existing structure of chain-based hash joins to factorize join results without requiring additional resources.

However, to repurpose the chains for factorization, all elements in a chain must have the same join key. This might not be the case naturally due to hash collisions, where as



(a) Hash table *without* key unique chains because of a hash collision: k_1 and k_2 share a chain.

(b) Hash table *with* key unique chains leading to k_1 and k_2 being in distinct chains.

Figure 4.1: Comparison between a hash table with and without chains that contain hash collisions

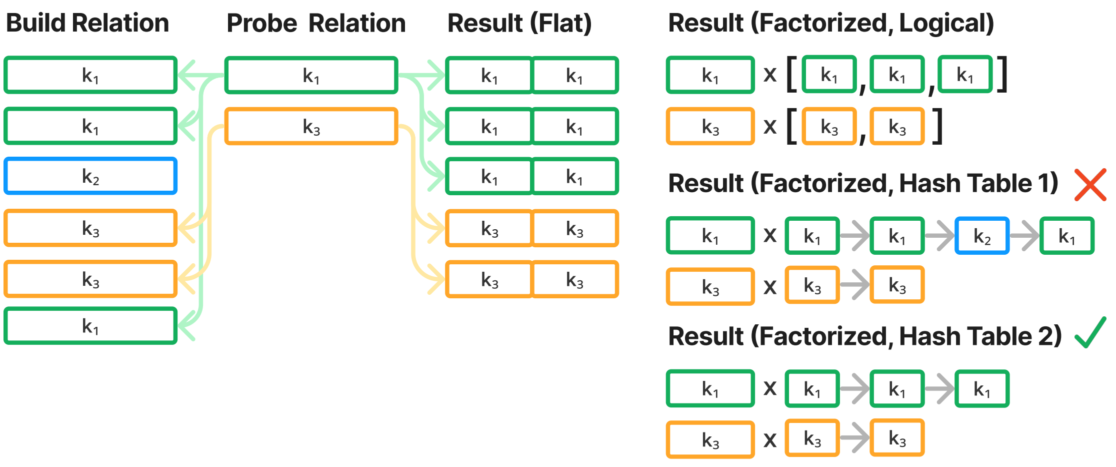

Build Relation
Probe Relation
Result (Flat)
Result (Factorized, Logical)
Result (Factorized, Hash Table 1)
Result (Factorized, Hash Table 2)
k1
k2
k3
x

Figure 4.2: Example of using hash table chains as factors. The first result shows the logical query result. The second result illustrates errors because of hash collisions within the chains. The third result shows the correct output when hash collisions are avoided.

discussed in Section 2.2.3 two distinct keys have the same hash value, mapping to the same chain. This is depicted in Figure 4.1a: Both k_1 and k_2 share the same chain. In contrast, the hash table in Figure 4.1b features a special handling strategy for collisions to ensure no collisions within a chain. This results in two distinct chains for items of k_1 and k_2.

Figure 4.2 illustrates what would happen if we used chains containing collisions using an example. Here, the goal is to join the relation $B = \{k_1, k_1, k_2, k_3, k_3, k_1\}$ on the relation $P = \{k_1.k_2\}$. For the sake of the example, B will be used as the build side, although it is the larger relation. The normal, user-facing flat result is listed in Equation (4.1), but it could also internally be represented in factorized as in Equation (4.2). This can be achieved by instead of emitting a tuple $k_{\text{probe}} \times k_{\text{build i}}$ for every key $k_{\text{build i}}$ found in the hash table, we collect them in a list L. The result would then be the multiplication of this list with the probing key but would remain in this compressed factorized form for further processing. This factorized logical version of the result is depicted in the top right corner of Figure 4.2.

$$\text{Flat: } R = (k_1 \times k_1) \vee (k_1 \times k_1) \vee (k_1 \times k_1) \vee (k_2 \times k_2) \vee (k_2 \times k_2) \tag{4.1}$$

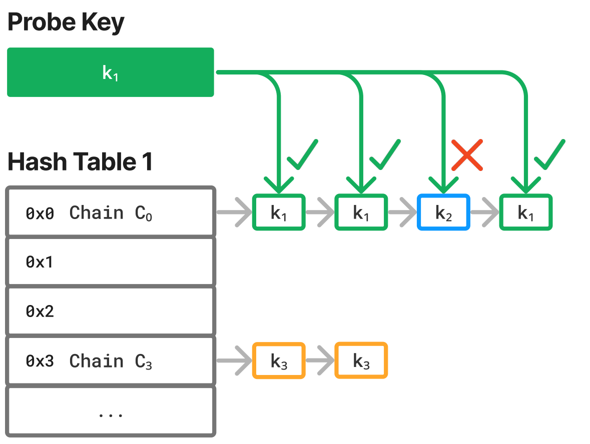
$$\text{Factorized: } R = k_1 \times (k_1 \vee k_1 \vee k_1) \vee k_2 \times (k_2 \vee k_2) \tag{4.2}$$

Assuming a hash operator would build a typical chaining hash table and assuming there is a hash collision between k_1 and k_2, resulting in $\texttt{hash(}k_1\texttt{)} = \texttt{hash(}k_2\texttt{)} = 0$, the hash table of the operator would be similar to the one depicted in Figure 4.3a. The problem is that the first slot of the hash table has tuples with both k_1 and k_2. Using hash table chains as lists for the factorization does not work in this case, as the chain corresponding to k_1 and k_2 can not be used as a factor without being first filtered only to contain k_1 or k_2. This is depicted in the right center of Equation (4.2), where the factor for k_1 would result in a wrong query result, as the tuple $k_1 \times k_2$ should not be in the query output.

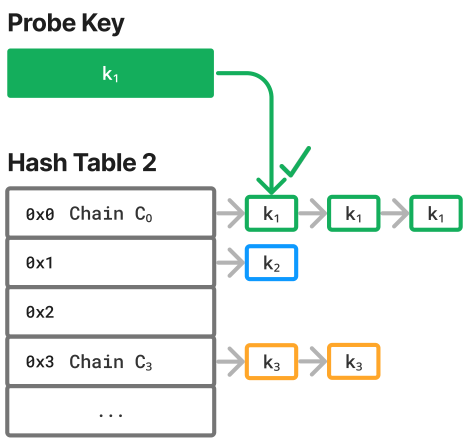
The second hash table solves this problem depicted in Figure 4.3b, which guarantees collision-free hashes. Here, the chains can directly be used as factors. Therefore, it is necessary to design a hash table to guarantee these collision-free chains.

4.2 Improving Performance With Collision-Free Chains

Ensuring that there are no hash collisions within a chain of the hash table can improve the performance of joins where there are duplicate join keys within the relation the hash table is built on. This is typical for many-to-many joins. Assuming a chain contains multiple keys, to scan for matches for a certain probe key, it must be compared against all the keys in the chain as every chain key, as each could be unequal to the one before due to a hash



(a) Without collision-free chains, to probe k_1, four comparisons are necessary



(b) With collision-free chains, comparing only the first element of a chain is enough

Figure 4.3: Comparison between the probing effort for hash tables with and without collision-free chains.

collision. If it could be ensured that all the keys in the chain are the same, an incoming probing key would only need to be compared once against the first elements of the chain. In this case, all chain keys after the first list can be considered matched without additional comparisons.

This process is depicted in Figure 4.3, calculating the join result $R = P \bowtie_{P.k=B.k} B$, building on relation B and probing with relation P: The first hash table is built on $B = \{k_1, k_1, k_2, k_1, k_3, k_3\}$.Figure 4.3a shows a hash table that uses chaining as a duplicate handling strategy but without collision-free chains. As visualized, there is a hash collision between k_1 and k_2, meaning that $\texttt{hash}(k_1) \mod B = \texttt{hash}(k_2) \mod B$, with B being the amount of buckets of the hash table. Therefore, when k_2 is inserted, it also gets appended to the slot with index 0. Thus, the chain in index 0 now consists of tuples with k_1 and k_2. To probe k_1, calculating its hash will lead to slot 0 with a chain $C_0 = \{k_1, k_1, k_2, k_1\}$. To get the join result $R_{k_1} \subseteq R$ for probing key k_1, all elements within chain C_0 must fulfill the join condition $P.k = B.k$. Therefore the result is $R_{k_1} = \{k_i \in C_0 \mid k_i = k_1\} = \{k_1, k_1, k_1\}$, where each tuple t in the result set matches the probing key k_1 based on the condition that $P.k = B.k.$, This means comparing each key within the chain against the incoming probing key.

Figure 4.3b shows a collision-free approach, which features an algorithm that guarantees that all keys within a chain are equal. How such a HT can be designed will be discussed in Section 4.3. In this case, if a chain is found where the first element matches with the key that is probed for, it can be guaranteed that all keys in the chain match. Therefore, as depicted in Figure 4.3b, only one comparison (green arrow) is needed to probe the key k_1 compared to the four comparisons of the previous approach.

This would mean that for many-to-many joins, the number of needed comparisons could be reduced, and therefore, the performance of the join algorithm improved. Such an improvement would especially benefit graph workloads, as many-to-many joins are common in this domain, as discussed in Section 2.3.2.

4.3 Approaches for Collision-Free Chains

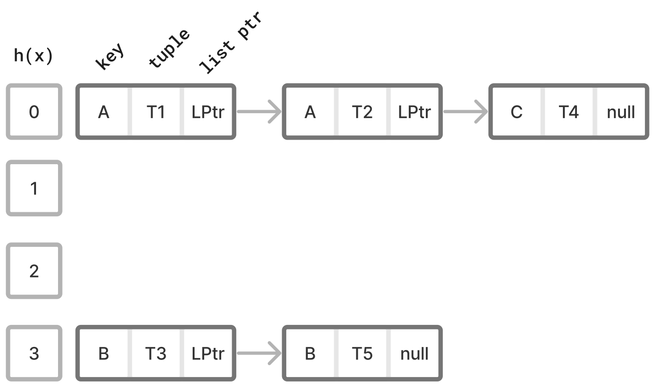
The following section will discuss two methods for achieving collision-free chains. These chains are necessary for their use as factors in factorized representations, as discussed in the section above, but also for enhancing performance when handling skewed data, particularly in many-to-many joins. After presenting the two approaches, they will be compared to decide which implementation to use for this project.

4.3.1 Nested Chaining

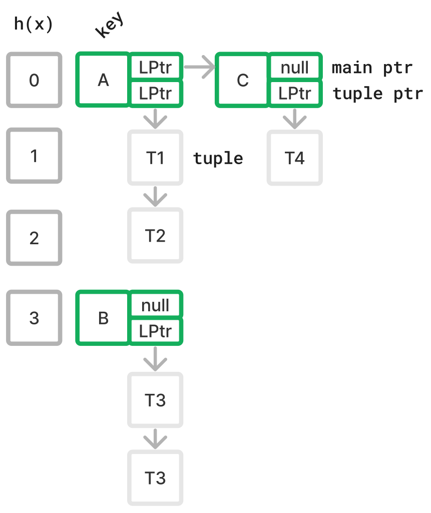
Flachs et al. [35] propose the "3D hash table" design which aims to reduce the length of collision chains by organizing each bucket hierarchically, with main nodes and sub-nodes, which also leads to collision-free sub-chains: Each main node is responsible for storing a distinct key and its associated value, while sub-nodes branching from a main node store only values related to that main key. This structure allows for shorter collision chains, resulting in fewer main memory accesses and less computational overhead during searches. While Memarzia et al. [64] proposed a similar approach, the following should mainly describe the approach from Flachs et al. [35].

During the insert operation, the algorithm checks if the key is already present in the chain of the main nodes. If it is, the value is added to the sub-chain of the existing main node. If the key is not present, a new main node is created. For lookups, the system checks the main nodes and their sub-chains to find the key, offering the option to either fully traverse the sub-chain for all associated values or return a nested tuple representing the packed search result. The authors showed that this hierarchical design manages collisions more efficiently and speeds up search operations.

Figure 4.4 illustrates a comparison between the traditional chaining method (Figure 4.4a) and the 3D hash table technique (Figure 4.4b). Each approach is represented by two hash table buckets constructed from the same dataset containing five rows. Each row contains a



(a) Default chaining with only one hierarchy of nodes (dark gray).



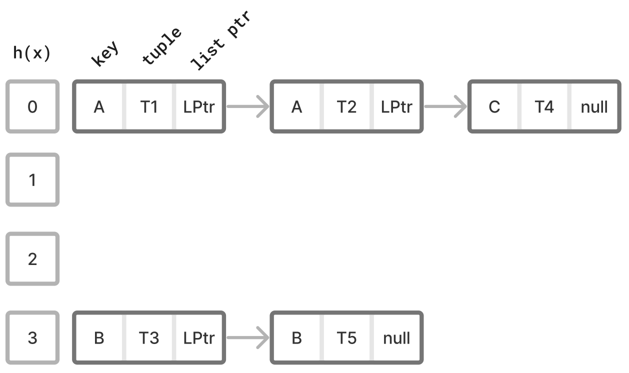
(b) 3D chaining with main nodes (green) and sub-nodes (light gray).

Figure 4.4: Comparison between default chained hash tables the combination of linear probing and chaining.

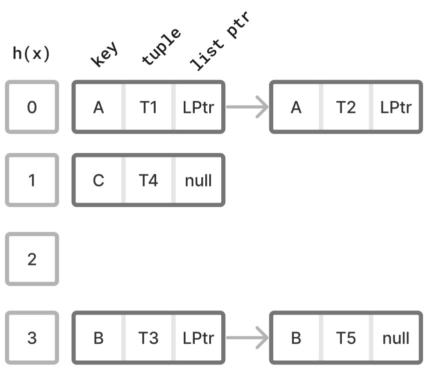
key (A, B, or C) and a set of column values T_1 to T_5. For instance, row 1 features the key A and the column set T_1. In a dataset on customers, the key A might represent a unique identifier, such as the customer's email address, with the tuple T_1 including further information like the customer's address and birthdate. To illustrate how the two approaches handle hash collisions, there are only four slots in the hash table, and the distinct keys A and C, $A \neq C$ collide in the first slot, as in this example hash$(A) =$ hash$(C) = 0$.

The structure of a traditional node in the standard chain is depicted in the first top-left element of the chain: Each node has a key which was used for the insertion. This payload tuple contains the other column values of the tuple, and a pointer to the next node. The hash collision is not handled as a special case but is just appended to the list of the first hash table. Therefore, the chain of slot zero now contains both nodes with key A and key C. If one wants to look up all matches for key C, it is necessary to traverse all elements of the chain of slot 0.

The 3D hash table handles these approaches differently: Using the chains of main nodes for distinct keys and the chain of sub-nodes for the tuples, there are now distinct chains for tuples of key A and tuples for key C. If the key C is probed now, only the main chain of nodes must be traversed until one finds a match (green). All sub-nodes (light gray) from this node will then be matched. This is especially important if, e.g., key A has a very long



(a) Default chaining with only one hierarchy of nodes (dark gray).



(b) Chaining with one hierarchy of nodes (dark gray) and linear probing.

Figure 4.5: Comparison between default chained hash tables and a combination of probing and chaining

chain, as compared to the previous approach, not all nodes of A must be scanned to reach the ones of C.

Flachs et at. [35] showed that the additional costs during the build and for the unnesting operation can be worth it due to a faster probe phase in many instances, especially when dealing with duplicate on the build side due to data skew. For a single key/foreign key join, the speedup factor of this method reaches up to 3.53 but also was slower in some scenarios. For many-to-many joins, the speedup factor was up to 5.67, with the 3D hash method always matching or exceeding the performance of the standard hash join.

4.3.2 Linear-Chained Hash Tables

Our new approach to guarantee chains with unique keys is to use a combination of linear probing and chaining. Here, the new key is compared to the appended first element of the chain. The new key will be appended to the chain if both are equal. If not, following the principle of linear probing, the next hash table slot is considered. If this is empty, the new value is appended to it; if not, it is again compared. We will refer to this approach as *Linear-Chained* hash tables.

This process is depicted in Figure 4.5 with the values equal to the experiment described in the above section. The linear-chained approach is depicted in Figure 4.5b: Here, when inserting the tuple T_4 using key C, calculating the hash of C leads to the hash table slot at the index 0. However, before inserting the key, it is compared against the first key in the chain, A. As $A \neq C$, following linear probing, the next slot of the following index is considered. As slot 1 is still empty, a new chain with tuples of key C is created. When probing for this key in a later stage, first, following linear probing, the first chain with a list of elements with key C is searched. If found, all its elements can be considered matches for key C. This, similar to the approach above, means a more complex build phase for the benefits of a more efficient probe phase.

4.3.3 Comparison of Nested Chaining and Linear Probing with Chaining

This section compares the nested chains approach proposed by Flachs et al. [35] with our Linear-Chained approach. The comparison focuses on two aspects: implementation effort and performance during both the hash table construction and probing phases. The results are summarized in Table 4.1.

Given that the implementation effort for integrating both approaches into DuckDB's Hash Join Operator is comparable, we decided to use the Linear-Chained approach over the nested chains approach. This choice was made due to the potentially higher performance in both the build and probe phases. This method more effectively reduces the number of chain traversals, which are costly because of cache misses resulting from nonlinear memory access. Additionally, linear probing can be further optimized by adding parts of the inserted key's hash as a *salt* to the hash table's slot, a technique discussed in Section 4.5.1.

However, the performance comparisons above are only theoretical. Implementing and benchmarking both approaches in practice would be an interesting direction for future work, but it is outside the scope of this project.

4.4 The DuckDB Hash Join Operator

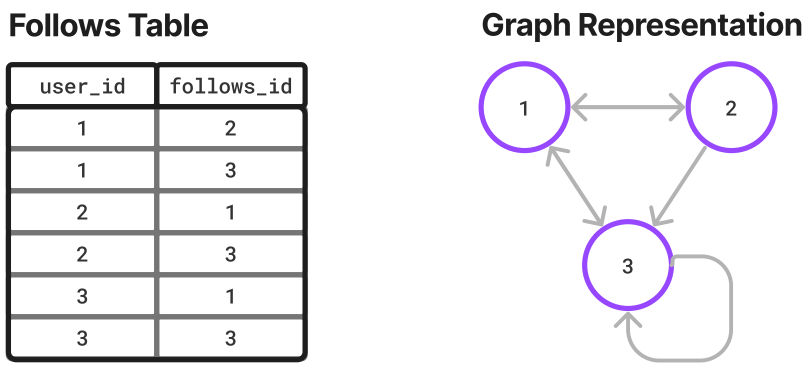
The following section will introduce the implementation of the DuckDB Hash Join Operator. As part of DuckDB's push-based execution engine, the Hash Join Operator is both a Sink and an operator: First, the complete hash table must be built on the right-hand side (sink, blocking), which then allows the probing with the left-hand side [25]. For more details, refer to Section 2.4. The sink phase is then again split into a materialization, which is then followed by the actual hash table population. To illustrate the existing implementation and the implementation proposed in the thesis on hand, the two approaches should be discussed using a simple example query in Listing 4.1. This query returns the followers of followers for all users in a social network for a given `follows` relation similar to one of a social network. Figure 4.6 shows both the `follows` table as well as a graph representation of the `follows` table, where each node is a user with its id and each row in the table

Category	**Nested Chains** HT	**Linear-Chained HT**
Implementation Effort - Building	*Higher* due to the need for a dual-chain hierarchy and dynamic memory allocation for main node chains. Complex to traverse and compare elements within chains of different hierarchies.	*Lower*, involves changes to the insertion algorithm using familiar linear probing techniques but can reuse existing chain structures.
Implementation Effort - Probing	*Lower* as the relevant sub-chain can be found using existing techniques. The sub-chain can then be easily traversed	*Higher*, add new linear probing algorithm, but this has to be implemented for building the hash table as well, so no big extra effort. When a matching chain is found, existing infrastructure can be used to produce the result.
Performance - Building	*Lower* due to potential memory allocation and cache misses from building and traversing main and sub-node chains.	*Higher*, can use existing hash table memory, with linear probing having fewer cache misses than traversing the chain of main nodes, as memory is accessed consecutively.
Performance - Probing	*Lower*, potential for cache misses when traversing main chains, lower table population due to unique chain heads outside the hash table, which reduces the number of hash collisions.	*Higher*, linear memory access patterns with linear probing may reduce cache misses, but hash collisions could occur more often as more slots will be occupied.

Table 4.1: Comparison of methods to achieve key-unique hash table chains. While both approaches require similar implementation effort, the linear probing approach will be more performant than having nested hierarchical chains.

```sql
SELECT
    f1.user_id AS 'User ID',
    f1.follows_id AS 'Friend of User ID',
    f2.follows_id AS 'Friend of Friend ID'
FROM follows f1
JOIN follows f2 ON f1.follows_id = f2.user_id;
```

Listing 4.1: Query returning the friends of friends for all users



user_id	follows_id
1	2
1	3
2	1
2	3
3	1
3	3

(a) Follows relation as table **(b)** Follows relation as graph

Figure 4.6: Follows relation used in the example query of Listing 4.1

represents a directed edge in the graph. Therefore, because of the first row $(1, 2)$, there is an arrow from node 1 to node 2, while because of the third row $(2, 1)$, there is also the arrow back. In the example join, the second follows relation (`f2`) relation will be used as the build side, leading to `user_id` being the build key. The result of the join R is therefore:

$$R = \texttt{Follows as f1} \bowtie_{\text{f1.follows\_id=f2.user\_id}} \texttt{Follows as f2} \tag{4.3}$$

This example will furthermore use the very naive hash function

$$\text{hash}(x) = x \mod 2 \tag{4.4}$$

mapping each key either to 0 or 1 to illustrate how the join implementation handles hash collisions.

The join plan for this query is split into two pipelines that need to be executed sequentially, as shown in Figure 4.7. The first pipeline is represented in green. It starts with a table scan as the data source and ends with a hash sink. Once the sink has consumed all data from the source, the hash table is built during the sink finalization step.

The second pipeline, shown in purple, is dependent on the completion of the first pipeline since it requires the fully constructed hash table for probing. Once the first pipeline is complete, the second pipeline begins, starting again with a table scan as the source. In this pipeline, the hash join functions as an operator, processing incoming data. The projection also operates as an operator. The final component in this pipeline is a sink, which returns the results to the user. As pipeline one blocks pipeline two, both pipelines cannot run concurrently. However, each pipeline can be executed concurrently with multiple threads.

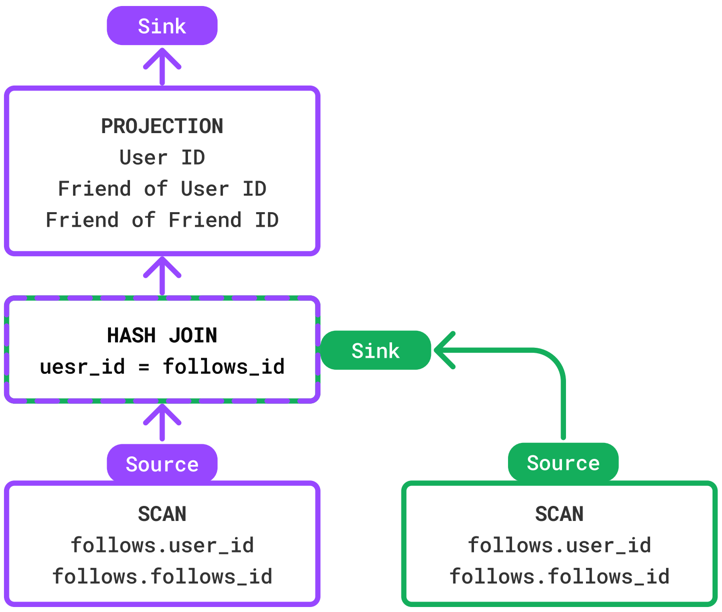
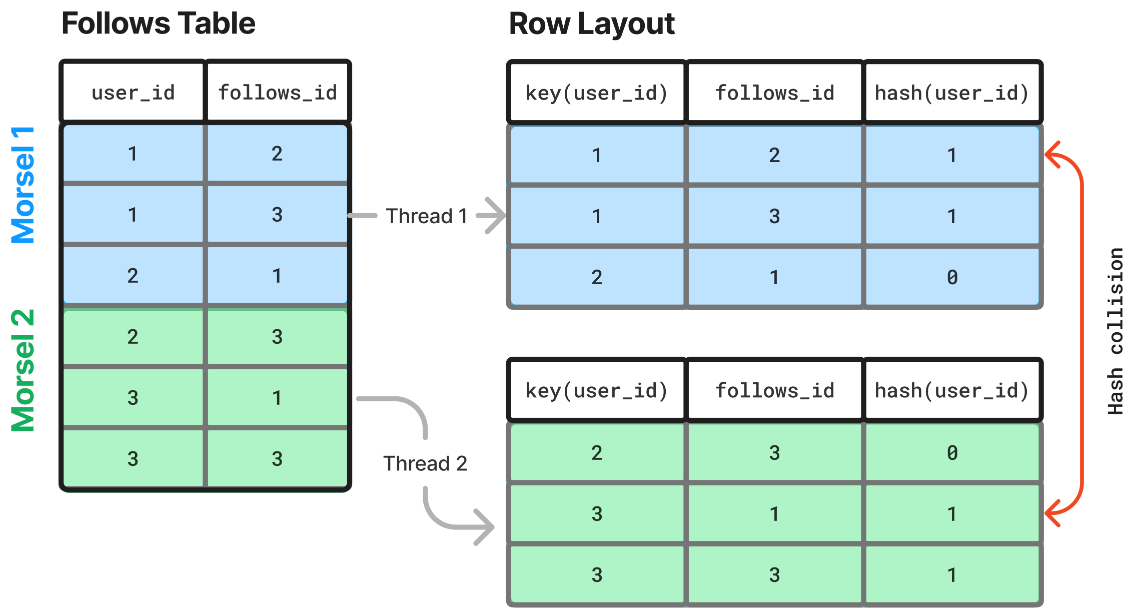


Figure 4.7: Pipelines and operators for the join in Listing 4.1: The green pipeline (scan → hash join build) blocks the purple pipeline (scan → probe → projection). Therefore, purple can only be executed after the green pipeline is completed.



user_id	follows_id
1	2
1	3
2	1
2	3
3	1
3	3

key(user_id)	follows_id	hash(user_id)
1	2	1
1	3	1
2	1	0

key(user_id)	follows_id	hash(user_id)
2	3	0
3	1	1
3	3	1

Figure 4.8: Sink phase for the join of Listing 4.1

4.4.1 Sink Phase: Materialization and Hash Calculation

During this phase, the build-side data chunks initially formatted in columns are converted to a row layout. The entire process is shown in Figure 4.8. Initially, the data is divided into morsels as described in Section 2.4 on morsel-driven parallelism. Separate threads then handle these morsels to achieve parallelism. For each incoming data row in column format, a new row is allocated within a temporary row store. The row format is organized as follows: firstly, the build keys are placed, which, for the example query, consists of only one key, `f2.user_id`. Following the key, the payload comprises columns to be emitted with the build key during probing. In this example, the payload contains only `f2.follows_id`, which will later be referred to as the "Friend of Friend ID".

Additionally, in each row, the key's hash value is computed and stored at the end of the row. The example defines the hash function as $\text{hash}(x) = x \mod 2$. In the example, the hash value for the first row is $\text{hash}(1) = 1$, while for the third row, with `f2.user_id` $= 2$, the hash is 0. A hash collision occurs in the last row, contained in morsel 2, where `f2.user_id` $= 3$ also results in a hash of 1, as $\text{hash}(1) = \text{hash}(3) = 1$. Thus, the first two rows and the last two rows share the same hash value despite having unique keys.

4.4.2 Sink Finalize: Building the Hash Table

The build phase starts after the whole build side has been materialized into the Row layout. In the pipelined execution model of DuckDB, it is triggered in the finalization step of the Sink. As this layout now contains all the rows, the number of rows the hash table needs to be built on is known. Therefore, the needed hash table buffer size can be determined ahead of build, so no costly resize of the hash table will be needed.

$$\text{NextPowerOfTwo}(x) = 2^{\lceil \log_2(x) \rceil}, \tag{4.5}$$

$$\text{Capacity}(n, \alpha) = \max\left(\text{NextPowerOfTwo}(\alpha \times n), 1024\right). \tag{4.6}$$

$$\text{with load factor } \alpha, \text{ number of rows } n$$

The needed capacity is calculated according to Equation (4.6): The load factor α balances the number of hash collisions with the memory usage of the hash table. With hash tables using chaining for duplicate resolution achieving good performance with a load factor $1 \leq \alpha \leq 3$ [62], DuckDB empirically chose a load factor of $\alpha = 2$. For the product of load factor and number of elements, the next larger power of two is then calculated. This makes sure that the capacity will be power of two, which enables an optimized hash table index calculating for a given hash using a bitmask:

$$\texttt{mask} = \texttt{capacity} - 1, \qquad e.g. \quad \begin{array}{rll} & 0100\ 0000_2 & \texttt{capacity} = 64_{10} \\ - & 0000\ 0001_2 & \texttt{dec} \quad\;\; = 1_{10} \\ \hline & 0011\ 1111_2 & \texttt{mask} \quad\; = 63_{10} \end{array} \tag{4.7}$$

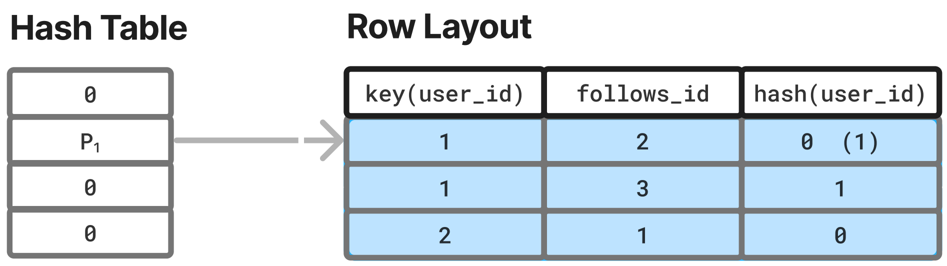
$$\texttt{index} = \texttt{hash} \wedge \texttt{mask}, \qquad e.g. \quad \begin{array}{rll} & 1011\ 0110_2 & \texttt{hash} \quad = 182_{10} \\ \wedge & 0011\ 1111_2 & \texttt{mask} \quad = 63_{10} \\ \hline & 0011\ 0110_2 & \texttt{index} \;\; = 54_{10} \end{array} \tag{4.8}$$

The binary representation of a power of two uniquely features a single active bit. Similarly, in base 10, a number that is a power of 10 is composed of a leading one followed by zeros. Subtracting one from these powers results in a number where all lower-order bits or digits become the maximum possible value for their position, and all higher-order bits or digits remain zero. For instance, subtracting one from 10^4(10,000) results in 9,999. In binary, subtracting 1 from 2^n(represented as $100\ldots0$ with n zeros) results in $11\ldots1$(with n ones). This process is also depicted in Equation (4.7).

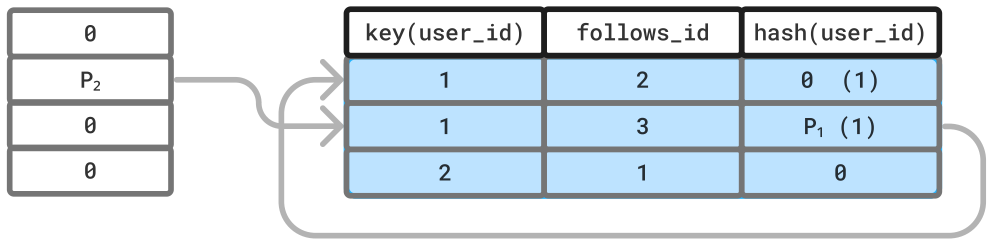
This mask can now be used to compute the index into a hash table of a given capacity: The slot index in the hash table can be calculated by a bitwise AND operation between the generated hash code and the mask. This approach ensures that the hash code is mapped within the capacity of the hash table. For instance, consider a hash table with a capacity of 64(which is 2^6), where the corresponding mask is 63(since $64 - 1 = 63$). Equation Equation (4.8) shows this for a hash value of 182_{10}. Directly using the hash value 182 to index the hash table would lead to an out-of-bounds error, as the table's array only extends to 63. However, by applying the mask with a bitwise AND operation, we compute an index of 54_{10}, which is within the range of the hash table.

Alternatively, one could use a modulo operation to align the hash value to the range of the hash table. While this would work, a classic modulo calculation implementation involves a division, which is considerably more expensive than multiplications. A single 32-bit division on a modern x64 processor has a throughput of one instruction every six cycles and a latency of 26 cycles. In comparison, multiplication has a throughput of one instruction per cycle and a latency of just three cycles [57]. Further alternatives for calculating the modulo efficiently involve a combination of multiplication and shift operation, which performs worse than the bit-mask but allows for arbitrary hash table sizes [88; 58].

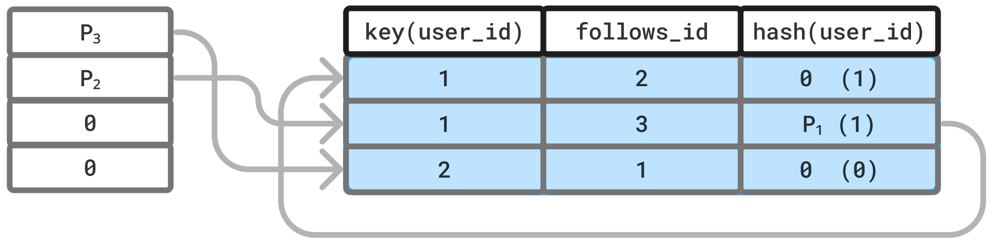
Reducing the hash values to indices within the hash table capacity, however, reduces its value range and power to distinct unique values, leading to more hash collisions that will lead to costly key comparisons. For example, while the hash in the example above



(a) Insertion of the first row in the empty slot with index 1



(b) Insertion of the second row in occupied slot 1, resolved by chaining



(c) Insertion of the third row in the empty slot 0

Figure 4.9: Insertion process for the first morsel of the example in Figure 4.6

originally has a value of a range of eight bits, which can represent 256 distinct values, now the index only has six "active" bits, resulting in 64 unique values. This leaves room for an optimization discussed in Section 4.5.1.

After allocating the hash table buffer and the index mask, the buffer must be populated. The chaining is implemented by storing pointers in the hash table buffer. Therefore, the buffer elements have a size of 64-bit. The insertion is depicted in figure Figure 4.9: For the first row, using the hash of 1 and applying the bitmask results in the slot s_1 at index 1. Therefore, a pointer to row 1 (P_1) which points to the beginning of row 1 is stored in slot 1. This step is depicted in Figure 4.9a. Figure 4.9b shows the insertion of the next row. As this also has the same join key, this row must somehow be stored to s_1. To chain the two rows, the pointer P_1, which was currently in the hash table buffer, is now stored in the place where the hash of row 2 was. This is possible, as both the pointers and the hash are 64-bit numbers, and after insertion, the row's hash is no longer needed. Then, the pointer to row two P_2 is stored in the slot s_1 instead.

```
 1 template <bool PARALLEL>
 2 static inline void InsertHashesLoop(
 3     atomic<data_ptr_t> ht_buffer[],  const data_ptr_t row_pointers[],
 4     const hash_t indices[], const idx_t count
 5 ) {
 6     for (idx_t i = 0; i < count; i++) {
 7         const auto index = indices[i];
 8         if (PARALLEL) {
 9             data_ptr_t head;
10             bool success;
11             do {
12                 head = ht_buffer[index];
13                 Store<data_ptr_t>(head, row_pointers[i] + pointer_offset);
14                 success = std::atomic_compare_exchange_weak(
15                     &ht_buffer[index],
16                     &head,
17                     row_pointers[i]
18                 );
19             } while (!success);
20         } else {
21             data_ptr_t head = ht_buffer[index];
22             Store<data_ptr_t>(head, row_pointers[i] + pointer_offset);
23             ht_buffer[index] = row_pointers[i];
24         }
25     }
26 }
```

Listing 4.2: Inserting a vector of row pointers to the hash table buffer. Instead of using a function argument to determine `parallel`, we use a template. This will allow the compiler to remove the parallel branch.

As the hash of a row is now used as the next pointer, it is also important to make it a null pointer if the row is the last row of a chain. As new elements are always added to the front of the chain, when starting a new chain by inserting the first element and adding a pointer to the buffer, the hash value of this element is overwritten with 0. In Figure 4.9, the hash values that have been overwritten are still visible in parentheses. As the hash table buffer is initialized with zeros, the insertions are the same for both empty and full slots: (1) Write the value of the current slot to the hash of the element to insert, which can be either zero or a pointer to another chain. (2) Write the pointer to the row to the hash table slot. The instructions are depicted in lines 24 to 27 of Listing 4.2.

This listing also shows how the implementation handles parallel insertion using a loop combined with an `atomic_compare_exchange` operation, ensuring the insertion is thread-safe. This technique prevents multiple threads from simultaneously writing to the same hash table slot. By using `atomic_compare_exchange`, only one thread can successfully write its value if the current slot matches the expected old value [12]. If the compare-and-exchange is successful, the element is correctly linked in the chain; if not, the operation is retried until it can be completed successfully.

In addition, Listing 4.2 is an example of the implementation of vectorized execution. Instead of only inserting one element at a time, a vector of `count` elements is inserted. Furthermore, the code is further optimized to reduce branching within loops: The bool `PARALLEL` is not a function parameter but a template parameter, allowing the compiler to optimize out the branch when the parallel mode is not enabled and vice versa. This reduces the runtime overhead and simplifies the code paths.

4.4.3 Operator Phase: Probe Phase

The probing phase of the hash table is implemented following DuckDB's framework for push-based execution: During this phase, in the `Execute` function, an operator is given a `DataChunk` to process. The first step is computing the hash for these keys and the index into the hash table. For the latter, the bitmask discussed above is used. The values of the slots at the indices are retrieved using these indices. These values can either be zero, indicating no matching key exists, or a pointer to a corresponding row based on the previously discussed row layout. The result of this operation is stored in a `PointerVector`. Simultaneously, a `SelectionVector` is constructed. It contains all entries in the `PointerVector` that contain a row pointer, signalling a match.

For the rows where a row pointer was found, the keys of the build and probe side first need to be compared to be sure that the match does not originate from a hash collision. For the matches, the build side tuples need to be concatenated with the probe side, again transforming the row format into a column format. This is done again in a vectorized manner: First, using the pointers to the matching rows, the probe side keys are compared against the keys in the row layout, retrievable using the row pointers. For each matching row, the elements for the build side are copied from the row layout into a result `DataChunk`.

Therefore, for each row of the probing `DataChunk`, the first row of the matching row layout is computed. However, as shown in Figure 4.9, these rows can be chained together if there is a key duplicate or a hash collision. Thus, for all the rows pointed in the `Pointer Vector`, the next element in the chain is loaded using the next pointer at the end of the row. The pointers are, therefore, advanced to the next pointer of the chain, which might be a null pointer if there is no next element. If there is at least one row where the pointer to the next element is not null, the process of matching and gathering needs to be repeated. This is done by returning “has more output” as a result of the `Execute` function, which leads to this function being called again with the same input. If there are no more elements to step to, the operator returns “need more import”, leading to it being called again with a new `DataChunk`.

4.5 Linear-Chained Hash Table Integration for DuckDB

With the functionality of the DuckDB Hash Join Operator outlined in the previous section, this chapter aims to describe the adaptations necessary to integrate collision-free chains into the DuckDB Hash Join Operator. The changes described here have been merged into DuckDB[1] and will become publicly available with DuckDB 1.1. As outlined in Section 4.3.3, this should be achieved using a combination of linear probing and chaining. The changes needed to implement collision-free chains only affect the hash table insertion and probing, but not the overall functionality of the Hash Join Operator, and will be outlined in the following.

4.5.1 Salted Linear Probing Optimization

While utilizing linear probing, due to the range of (nested) types that can be potential join keys, they are not stored directly in the hash table. As outlined in Section 4.4.2, the hash table stores pointers to the rows rather than the rows themselves. This reduces the memory area exposed to random access and decreases overall memory consumption [52]. Here, we can apply an optimization: On 64-bit CPU architectures, pointers are stored in 64 bits; however, typically, only the lower 48 bits are actively utilized, already addressing up to approximately 281 terabytes. This enables repurposing the upper 16 bits of each pointer to store the `salt`, which consists of the upper bits of the hash associated with the tuple.

The initial position within the hash table is determined by the lower bits of the hash. If a collision check is necessary, before a pointer is dereferenced to compare the rows’ join keys, the salt stored in the pointer is checked. A matching salt significantly reduces the likelihood of unnecessary comparisons of join keys—by a factor of $2^{16} = 65,536$. Thus, if the salts do not match, the keys also must not match, and the search proceeds to the next

[1]`https://github.com/duckdb/duckdb/pull/11472`

slot in the hash table. However, if the salt matches, further comparison of the keys is still required.

4.5.2 Changes Made to the Build Phase

As before, the index of an element is retrieved using a bit-mask in combination with the hash. In the previous implementation, a row was added to a chain of a hash table slot, no matter whether the slot already had a chain or whether the slot was empty. These two cases now need to be handled differently than before:

If a slot is empty, the pointer to the new chain head and the salt are inserted into the slot. the salt is calculated using s bitwise AND operation with the hash and a mask for the upper 16-bit, further referred to as `SALT_MASK` and defined as `0xFFFF000000000000`. The salt is then combined with the row pointer.

$$\begin{aligned}
\texttt{SALT} &= \texttt{HASH} \ \wedge \ \texttt{SALT\_MASK} && (4.9)\\
\texttt{SLOT\_VALUE} &= \texttt{POINTER} \vee (\texttt{SALT} \ \wedge \ \text{SALT\_MASK}) && (4.10)\\
\texttt{POINTER} &= \texttt{SLOT\_VALUE} \ \wedge \ \texttt{POINTER\_MASK} && (4.11)\\
\texttt{SALT} &= \texttt{SLOT\_VALUE} \ \wedge \ \texttt{SALT\_MASK} && (4.12)
\end{aligned}$$

If a slot is already occupied, the new element can only be appended if the keys of the element to insert and the elements in the existing chain match. First, the salts are compared. If those are equal, the keys are compared by following the pointer in the slot. Retrieving the pointer from a hash table entry is done by masking the stored value with `POINTER_MASK` (defined as `0x0000FFFFFFFFFFFF`), which extracts the lower 48 bits.

In the case of multi-threaded insertion, ensuring key unique chains requires additional checks. For example, if the compare and swap operation described in Listing 4.2, line 23, fails when trying to insert into a previously empty slot, now before trying to insert the element again, the keys need to be compared. This is because it could be that the other chain that was concurrently added might not share the same key as the other element. If the insertion fails while adding an element to an existing chain, it can just be repeated, as it was already ensured previously that the keys match.

4.5.3 Changes Made to the Probe Phase

To probe the hash table with a `DataChunk`, for each row in the vector, the hash and the index are calculated to match a slot to each index. If the entry is empty, then there is no match. For all keys that result in a hit, and if the salt comparison is enabled, the salt is calculated using their hash. Then, for each key to be probed, the salt of the respective slot is checked. If the salt does not match, the search continues to the next slot until either an empty slot is found or one where the salt does match. In the case of finding an empty slot, there is no match and the key is not processed any further. If a salt match is found, the probe key is compared against the key of the first element in the list of the hash table slot. Should the comparison be successful, a pointer to the list of the entries is returned, and the probe key is added to the return `SelectionVector`. If there is no match, the next slot is checked, and the loop starts again. We could further optimze this by using one bit in the salt to mark if there was a hash collision during the build process. If this is not the case and the salt does not match, we could immediately stop the probing, making selective joins more efficient.

The probing can operate in two modes: using salt or not using salt. This was added because for smaller hash tables, the likelihood of both hash collisions for keys and cache misses while comparing the incoming keys with the rows inside the hash table is lower. For instance, in a hash table with only five distinct keys but a million probe tuples, the salt comparison will always be correct. The current criterion for not using salt is if the hash table size is 8192 entries or smaller.

As the overall logic of the Probing is now more complex than in the previous implementation (calculating and comparing salt, comparing keys), one big optimization was to hide this logic from keys where there are no matches in the hash table. This is why, first, the incoming `DataChunk` is filtered for all keys that have a match in the hash table. Only for rows where there are matches, the salt is calculated.

The probing phase of the join hash table returns a vector with pointers to matching chains. Now, where the previous implementation had to walk the chains for each row and compare all the entries, in the new implementation it is known that all elements will match the row. For illustration, see again Figure 4.3.

Another optimization is that the new implementation tracks whether there are chains with more than one element. If there are no chains with more than one element, the pointer advancement described in Section 4.4.3 is not needed.

4.6 Evaluation of the Linear-Chained Hash Table

The modifications and changes to the hash table discussed will now be assessed using benchmarks that simulate real-world workloads. The setup of these benchmarks, as well as the hardware used, will be introduced in Section 4.6.1, while the actual benchmark results will be presented in Section 4.6.2.

4.6.1 Experiment Setup

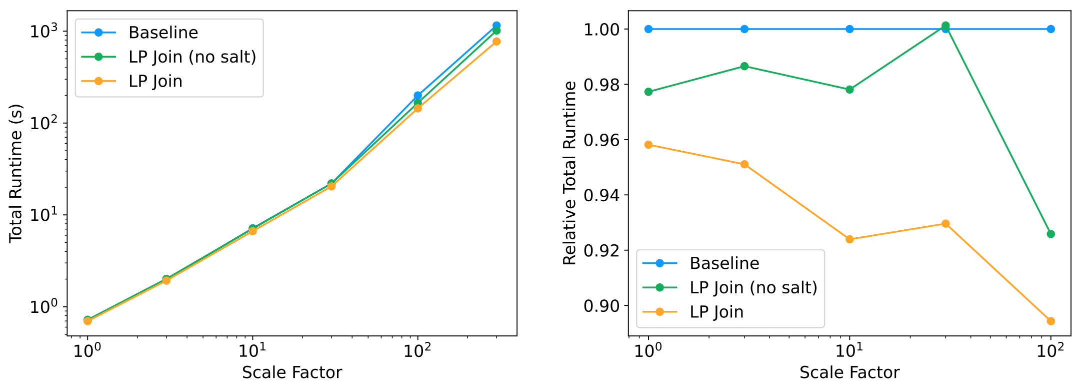
As DuckDB is mainly used on high-end end-user devices, the evaluation was performed on a MacBook Pro with an M1 chip using 6 threads.

As evaluation workload, the TPC-H and the TPC-DS benchmark were used. The Transaction Processing Performance Council (TPC) is a non-profit organization that develops performance benchmarks for various data processing systems [90]. Among its benchmarks, TPC-H and TPC-DS are designed for evaluating decision support systems. TPC-H [97] assesses the capability of a system to manage complex queries and data modifications typical of business environments, while TPC-DS [96] offers a more detailed evaluation environment by modeling decision support systems with different queries and data maintenance operations.

TPC-D, along with its successors TPC-H and TPC-R, originally utilized a 3rd Normal Form (3NF) schema which was discussed in Section 2.1.1. However, over time, the industry trend has shifted toward adopting star schema approaches. In response to these changes, TPC-DS was developed using a multiple snowflake schema, which represents a hybrid approach that combines elements of both 3NF and a pure star schema [72].

For DuckDB as well as for the project on hand, it is important to add that the benchmark results discussed here are not official TPC benchmark results; they utilize datasets and certain queries from the TPC-H and the TPC-DS benchmark but do not include complete workloads such as updates to the data [17]. The benchmarks can be run using predefined database sizes called ”scale factors.” Each scale factor represents the raw data size of the data warehouse, meaning that a scale factor of 1 means 1GB of uncompressed data. To give a reference for how much data this is, this corresponds to 1,500,000 rows in the `orders` table for the TPC-H benchmark For the project on hand, the benchmarks were run using the scale factors 1, 3, 10, 30, and 100.

For both the TPC-H and TPC-DS benchmark, we compare three different versions of DuckDB: The first is the baseline version, which is the state of DuckDB before the changes to the hash table. The second version features the new hash table, which combines both chaining and probing. This version will be referred to as Linear probing (LP) Join version. To analyze the effectiveness of the salt optimization for cash misses, the third version is based on the LP Join version but does not feature the salt optimization.



(a) Absolute total runtime (lower is better) **(b)** Runtime relative to the baseline (lower is better)

Figure 4.10: TPC-H Benchmark: Absolute and relative total runtime

Random variances, such as background tasks slowing execution speed or effects of pre-cached data, can influence benchmark results during query execution. To mitigate these influences, each query in the benchmark is run five times. The final runtime of the query is then determined by the median of these five runtimes. We chose the median over the mean is it is less responsive to outliers.

4.6.2 Results & Discussion

The results for the total runtime of all queries of the TPC-H queries can be seen in Figure 4.10. The figure shows both the relative and absolute runtime of the whole benchmark to complete. Both the runtime on the y-axis and the scale factor are in the log scale. Looking at the absolute runtime, it becomes apparent that all versions are similar in performance, as the lines are close together. However, we can see that the LP Join has a lower runtime than the baseline, especially towards higher scale factors.

The absolute runtimes per scale factor are also listed in Table 4.2: We can see that for both versions the runtime increases with the scale factor, but the difference between the runtime gets larger: While the baseline takes 0.93 seconds to complete the whole benchmark, the new implementation without salt took 0.91 seconds, and with salt, the runtime was reduced to 0.89 seconds. With higher scale factors we see the runtime difference between the versions getting larger: While the optimized version of the new join takes only 147.91 seconds to compute all TPC-H queries, the baseline version needs 165.39 seconds, therefore being 14.48 seconds slower.

The relative runtime offers a better understanding of the improvements of the three

System	SF 1	SF 3	SF 10	SF 30	SF 100
Baseline	0.93	2.33	8.00	28.06	165.39
LP Join (no salt)	0.91	2.30	7.82	28.09	153.12
LP Join	0.89	2.22	7.39	26.08	147.91

Table 4.2: TPC-H - Total runtime in seconds (lower is better)

System	SF 1	SF 3	SF 10	SF 30	SF 100
LP Join (no salt)	97.73%	98.66%	97.81%	100.13%	92.58%
LP Join	95.82%	95.11%	92.39%	92.96%	89.43%

Table 4.3: TPC-H - Relative runtime over baseline (lower is better)

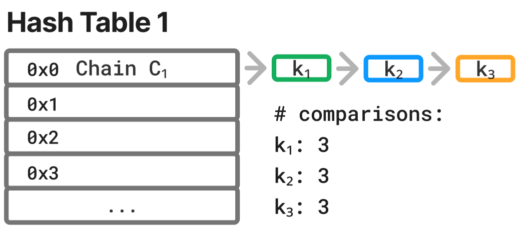
versions: in Figure 4.10b we can see the relative runtime over the baseline per version. The relative runtime r_{rel} is computed as

$$r_{rel,i} = \frac{r_i}{r_0} \tag{4.13}$$

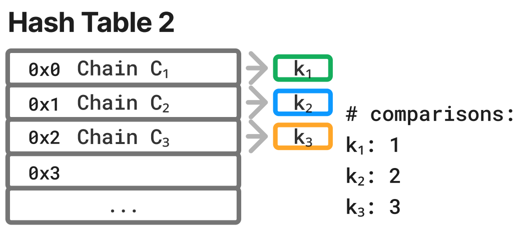
where r_0 is the absolute total duration the baseline version took to complete and r_i is the duration the version to compare took to complete. For example, if the baseline version took 2 seconds to complete and the new version manages to run the same workload to compute in 1 second, the relative runtime will be $r_{rel,i} = \frac{1}{2} = 0.5$. Therefore, a lower relative runtime indicates a better-performing system.

Figure 4.10b compares the relative runtime of the new join implementation with and without the salt optimization against the baseline. We can see that for the relative runtime of the two new implementations decreases with the scale factor. This means that the new implementations become more effective the larger the data size is. As the new join implementation strength is handling a lot of duplicates of the same key, assuming larger scale factors add more data for the same domain like adding more buy records for the same customer instead of new customers, the new algorithm becomes more effective.

Analyzing the orders table of the TPC-H benchmark proves this assumption to be false: There are 1,500,000 orders on scale factor one from 99,996 distinct customers, and the number of average orders per customer is around 15. This is the same for scale factors 10 and 100, with 100 having 9999832 distinct customers in 150000000 orders, again resulting in an average of 15 customers per order. Unlike in this benchmark, in a real-world scenario, more data could also mean more data of the same scope, which could make the new



(a) For each probe key all elements have to be scanned.



(b) For each probe key we need to scan only until a matching chain key is found

Figure 4.11: Comparison between the new implementation of the hash table against the table before when building on keys with no duplicates

implementation more performant. However, this is not the explanation for the relation of speedup and scale factor in this case.

An alternative reason for the performance gains could be that the linear probing enables a faster probing for the hash table without having duplicates by better-separating hash collisions. In the version before, one chain could contain hash collisions at every point in the chain. This means that for an incoming tuple, all elements of the chain had to be scanned, even when joining on a key with no real duplicates. In the new implementation, when joining with a key with no duplicates, the new implementation can still be more performant when being probed. The reason for this is depicted in Figure 4.11. Assuming we build the hash table on keys k_1, k_2, k_3 with the three keys being distinct but colliding on the same hash. With only chaining, all three keys will end up in the same chain. When we now probe for k_2, we have to compare our probing key against all keys in the chain, as all of the keys in the chain can either be hash collisions or actual matches. The same holds for the other two keys (see Figure 4.11b). Therefore we need nine comparisons in total.

With the new implementation, the keys get split into separate chains. This is depicted in Figure 4.11b. If we now probe for k_2, we first arrive at the chain of k_1, so we need to compare the two keys. As they do not match the next bucket is considered, following the principles of linear probing. Comparing the probing key with the first element of the chain of this bucket, we find that the keys match, therefore we can stop the search and consider all elements of this chain matches. For k_1, we even only need one comparison as the first chain already matches, while for k_3 still three comparisons are necessary. In total now only 6 instead of 9 comparisons are necessary.

However, the new implementation requires comparisons during the build phase. In the example of Figure 4.11b, we would need three comparisons during the build phase: When inserting k_1, the hash table is still empty, so no comparisons are necessary. If we then insert k_2, we first end up at k_1, so after comparing the keys we step to the next. Therefore we need one comparison. For k_3 we need two comparisons following the same principle, so the total amount of comparisons for the build phase is three for the new implementation and zero for the old.

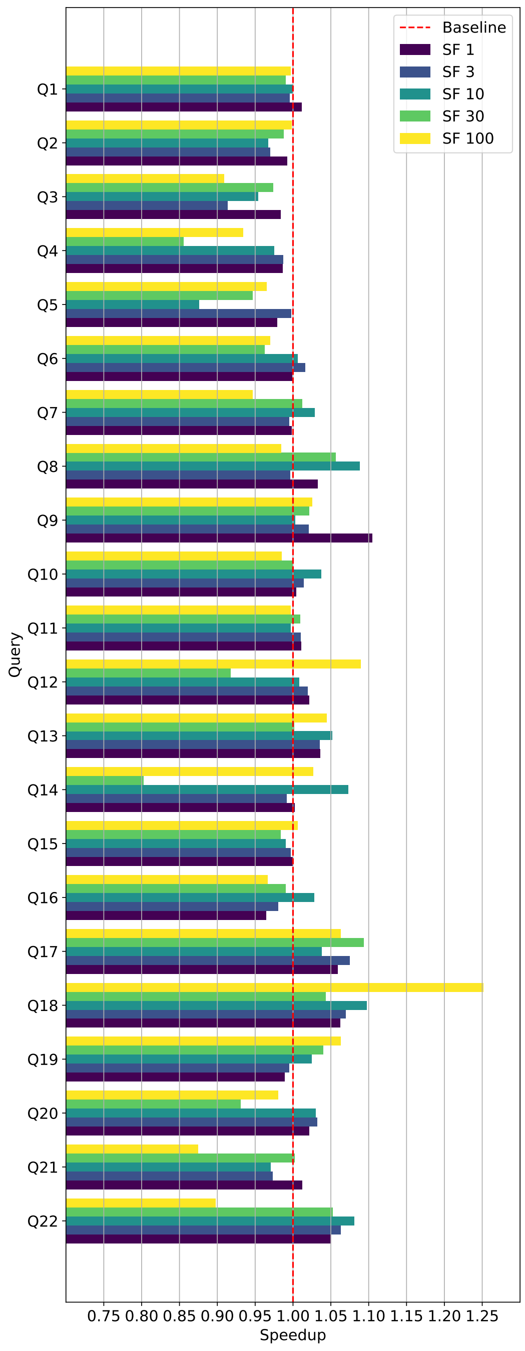
Summing the probe phase and the build phase comparisons together, we end up with 9 comparisons for both implementations. However, it must be noted that usually, the probe side is much larger than the build side, which justifies additional work during the build. For example, assuming the probe side is ten times larger than the build side, we would end up with 90 vs 63 comparisons, so this would make the new implementation more efficient.

Looking at the salt optimization, the yellow line in Figure 4.10b shows that the version using salt optimization has a lower relative runtime compared to the baseline as the scale factor increases. This indicates that the new implementation becomes more efficient with larger data sets. This behavior is expected because salt optimization primarily addresses cache misses, which become more problematic as data sizes increase beyond the cache's capacity. Table 4.3 shows that already with a scale factor of 1, the relative runtime is only at 95.82%. The performance gains increase to a relative runtime of 89.43%, which translates into a speedup of 12%.

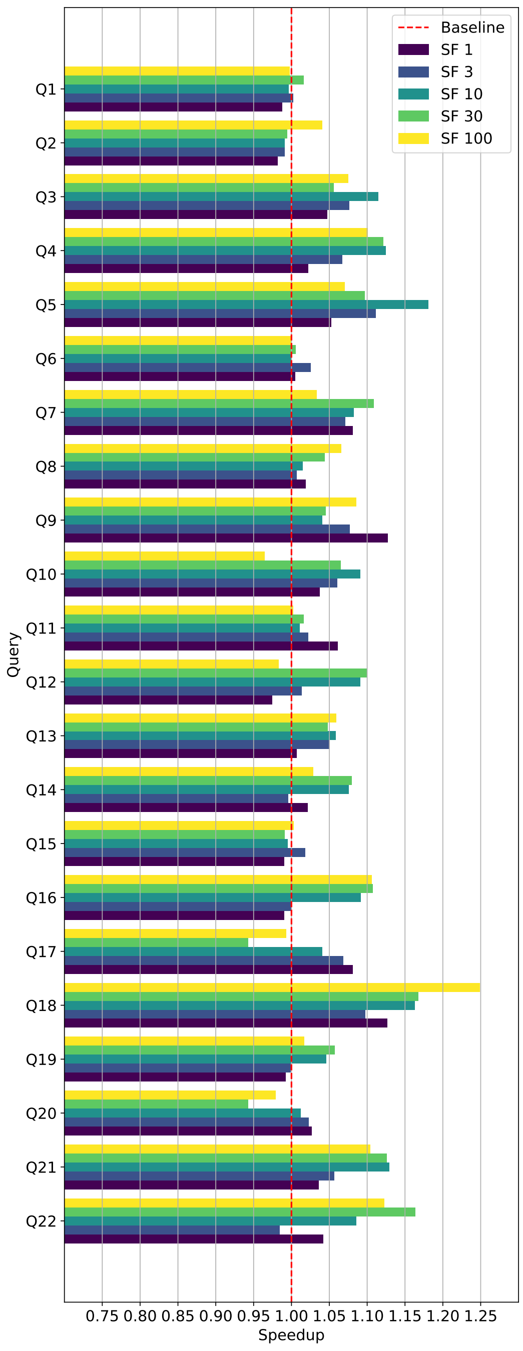
Figure Figure 4.12 provides a more detailed on of the relative runtime over the baseline per query and a scale factor: for each of the 22 queries of the TPC-H, benchmark $Q1 \dots Q2$ the speedup over the base implementation per scale factor is plotted. The runtime per scale factor is indicated by the color of the bar, with purple corresponding to a scale factor of 1 and yellow to a scale factor of 100. As the figure shows the speedup rather than the relative runtime, higher numbers are more favorable. The baseline speedup of 1 is indicated with the red dotted line, so a bar left of this line indicates a decline in performance, while a bar right of it indicates an improvement.

The plot shows that for $Q18$ we get performance increases for both the version not using and using salt. These get higher the higher the scale factor is. Analyzing $Q18$'s (see Listing 4.3) query plan, we can see a larger join. This originates from the `o_orderkey = l_orderkey` joining the `orders` table with the `lineitem` table, which accounts for 40% of the runtime. The probing side here is ten times larger than the build side. While the hash table for this join is built on a primary key, as previously mentioned, the new implementation remains superior due to the offload of work to the (smaller) build side.

The second experiment compared the performance of the different versions for the TPC-DS benchmark. The results of the absolute and relative runtime of all queries are



(a) Linear Probing, no salt (higher is better)

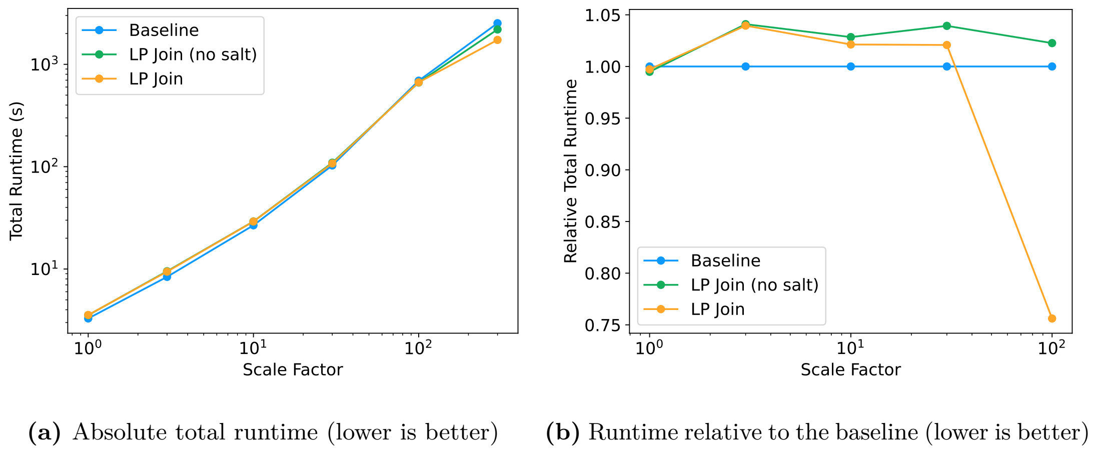


(b) Linear Probing (higher is better)

Figure 4.12: TPC-H Benchmark: Speedup per query and scale factor

```sql
SELECT c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice, SUM(l_quantity
    )
FROM customer, orders, lineitem
WHERE o_orderkey IN (
    SELECT l_orderkey FROM lineitem GROUP BY l_orderkey HAVING SUM(l_quantity)
    > 300
) AND c_custkey = o_custkey AND o_orderkey = l_orderkey
GROUP BY c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice
ORDER BY o_totalprice DESC, o_orderdate
LIMIT 100;
```

Listing 4.3: TPC-H query 18



(a) Absolute total runtime (lower is better) **(b)** Runtime relative to the baseline (lower is better)

Figure 4.13: TPC-DS Benchmark: Absolute and relative total runtime

System	SF 1	SF 3	SF 10	SF 30	SF 100
Baseline	4.19	10.23	31.18	111.71	467.10
LP Join (no salt)	4.17	10.65	32.07	116.10	477.66
LP Join	4.18	10.63	31.85	114.04	353.19

Table 4.4: TPC-DS - Total runtime in seconds (lower is better)

System	SF 1	SF 3	SF 10	SF 30	SF 100
LP Join (no salt)	99.49%	104.09%	102.84%	103.93%	102.26%
LP Join	99.73%	103.95%	102.13%	102.08%	75.61%

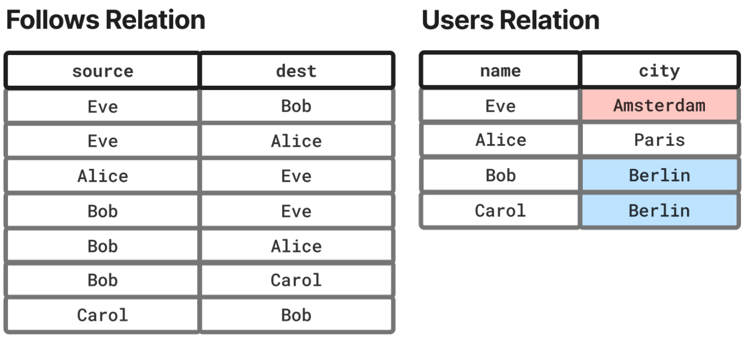
Table 4.5: TPC-DS - Relative runtime over baseline (lower is better)

depicted in Figure 4.13. Here we can see, that for most of the scale factors, the baseline implementation is better than the new version. The linear probing joined with the salt optimization has its worst performance at scale-factor 3, with a performance overhead of 3.95% leading to an additional absolute runtime of 400ms. However, for scale-factor 100 the new implementation is drastically better, featuring a relative runtime of only 75.61%, which saves 113.91 seconds of the total query runtime (see Table 4.4 and Table 4.5). This proves the point that salt optimization is reducing the number of cash misses, which occur more often on larger scale factors, as intermediate structures like the hash table array do not fit into the CPU cache anymore.

Additionally, it becomes apparent, that the version not using the salt is slower for all cases. This is because of the star schema of the TPC-DS benchmark, with no joins having a lot of duplicates on the build side.

4.7 Emission of Factorized Intermediate Results

This section will cover how we can use the linear-chained hash table in our intermediate results. The creation of these chains has been discussed in Section 4.1.



Follows Relation

source	dest
Eve	Bob
Eve	Alice
Alice	Eve
Bob	Eve
Bob	Alice
Bob	Carol
Carol	Bob

Users Relation

name	city
Eve	Amsterdam
Alice	Paris
Bob	Berlin
Carol	Berlin

Figure 4.14: Relations for the factorization example: The User information has information on the city of birth of a user, while the `follows` relations represent a many-to-many relation between users.

To illustrate the explanation, the query depicted in Listing 4.4 will be used: It describes a query on a social network-like database schema featuring one `users` relation containing a user's name and place of birth as well as a `follows` relation representing a many-to-many relation between users. The two tables are depicted in Figure 4.14. The query returns the number of 2nd-degree followers of all users grouped by the city.

Analyzing the query plan and the cardinality of its intermediate results shows that this query has an intermediate result that is larger than both the normalized input relations

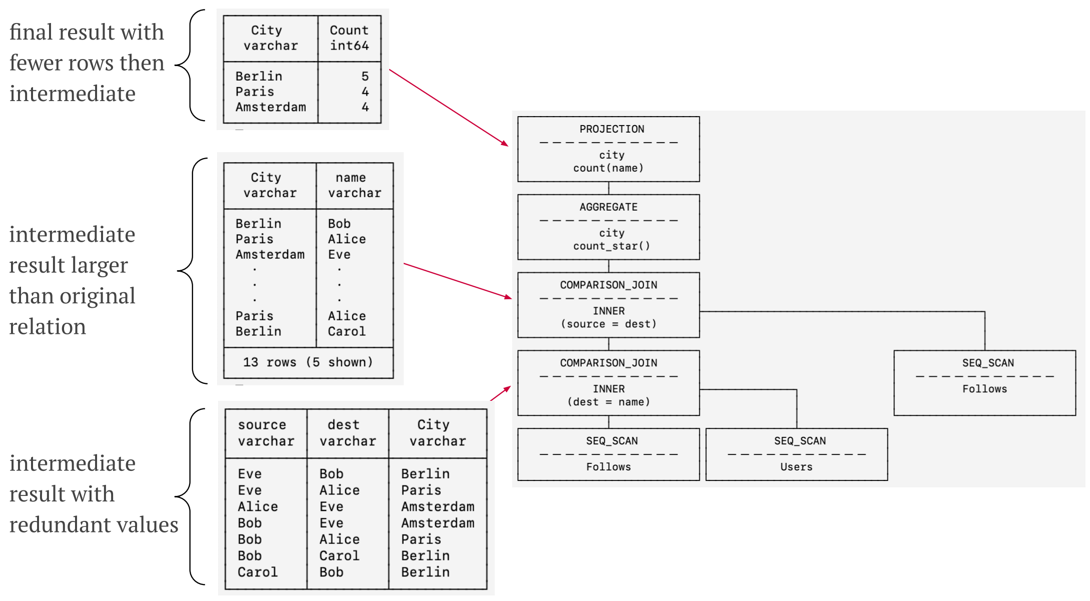
```sql
SELECT
    users.city AS 'City',
    COUNT(users.name) AS 'Count'
FROM
    Follows as F1
JOIN
    Follows as F2 ON F1.dest = F2.source
JOIN
    Users ON F2.dest = Users.name
GROUP BY
    Users.city;
```

Listing 4.4: Query returning the number of 2nd degree followers grouped by place of birth

and the final query result. The query plan and its intermediates are depicted in Figure 4.15. It is executed by joining the `follows` table on the `users` table using the `users` table as the build side and the `follows` table as the probe side for the hash join operator. This intermediate is already larger in terms of columns but not in terms of rows, as for each user the corresponding city is now included. Therefore this intermediate result has more duplicates than its normalized origins. It will further be referred to as `intermediate 1`. In this case, the hash table is built on the primary key of users, the user name. Therefore there are no duplicates in the keys of the build side, so all the chains in the hash table will be of length 1. Therefore, in this join, there is no potential for factorization.

The second operator is another comparison join which joins the `follower` relation with `intermediate 1`. Here, the `Follows` relation is chosen as the build side of the hash join, and the `intermediate 1` is used as the probing side. In this case, the hash table is built on `follows.dest`, a foreign key. Therefore, there are duplicates in the build keys which leads to chains being longer than one. This also becomes apparent when looking at the cardinality of the result of the second join, `intermediate 2`. While both the probe side and the build side have a row count of 7, the intermediate result now has a row count of 13. This is because there are multiple matches for some probing keys. For example, as there are two matches for the probe key `Bob` (the first and the last row of the `follows` relation), for the 4th probing row there will be two matches in the hash table, leading to two rows in the intermediate result.

The last operator of the query plan is an aggregate, grouping all tuples of the `intermediate 2` by the city. Therefore, the result cardinality is three, as there are only three distinct cities in the dataset.



City varchar	Count int64
Berlin	5
Paris	4
Amsterdam	4

City varchar	name varchar
Berlin	Bob
Paris	Alice
Amsterdam	Eve
.	.
.	.
.	.
Paris	Alice
Berlin	Carol
13 rows (5 shown)	

source varchar	dest varchar	City varchar
Eve	Bob	Berlin
Eve	Alice	Paris
Alice	Eve	Amsterdam
Bob	Eve	Amsterdam
Bob	Alice	Paris
Bob	Carol	Berlin
Carol	Bob	Berlin

Figure 4.15: Query plan with intermediate results of the query depicted in Listing 4.4: The intermediate results grow larger than both the input and output relation

A part of the hash table that is built during the sink phase of the 2nd hash join operation is depicted in Figure 4.16: We can see that for the build key `Eve` of the `follows` relation there is a chain containing two elements. These two correspond to the 3rd and 4th row in the `follows` relation. Additionally, the figure shows the first two rows of the probing relation, `intermediate 1`.

In a standard join implementation, for each of the two probing rows of Figure 4.16, a new tuple would have been created. The result of this is shown in Figure 4.16a, which shows the four resulting rows. This approach is the reason for the explosiveness of intermediate results for many-to-many joins, especially in graph use cases: Assume `Eve` would be followed by 100 persons. This would mean that for each match on the probe side, 100 rows need to be added. Creating these rows means duplicates for the probe side for each build match as well as duplicates of the build side tuples for each probe hit. In Figure 4.16a this becomes apparent: Both probing tuples and building tuples had to be copied, resulting in four rows.

The alternative to this approach is to not create one new row for each element of the hash table but to transfer the hits of the build side to a list and use these lists as a factor for a factorized representation. This is depicted in Figure 4.16b and enables us to not create duplicates of the probing side and also keeps the number of rows smaller or equal to the probing side. In the theory of factorization, this would correspond to an

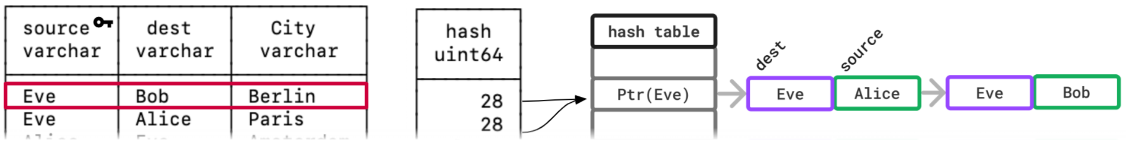


Figure 4.16: The hash table for the 2nd join of the join plan is depicted in Figure 4.15, as well as the first two rows of the intermediate result, represent the probe side on the left

source varchar	dest varchar	city varchar	source varchar
Eve	Bob	Berlin	Alice
Eve	Bob	Berlin	Bob
Eve	Alice	Paris	Alice
Eve	Alice	Paris	Bob

(a) Flat result consisting of four rows and having duplicates on both the probing and build side

source varchar	dest varchar	city varchar	source varchar[]
Eve	Bob	Berlin	[Alice, Bob]
Eve	Alice	Paris	[Alice, Bob]

(b) Factorized emission using materialized lists as factors, which corresponds to an f-representation. Only the build side chains need to be duplicated for every hit.

source varchar	dest varchar	city varchar	source varchar
Eve	Bob	Berlin	0x7ffee2c3a9bc
Eve	Alice	Paris	0x7ffee2c3a9bc

(c) Factorized emission using pointers to the hash table chains, which corresponds to a d-representation. Neither the probe nor the build side tuples need to be copied.

Figure 4.17: Different methods of producing flat or factorized join results for the two probing rows in Figure 4.16

f-representation [80]. However, in this approach, we still need to materialize one chain for each probing match: For the hash table chain, we still make two copies of the list [`Alice`, `Bob`]. While this approach manages to ensure that the cardinality of the join result is never bigger than the cardinality of the probe side, materializing all the lists is still expensive. Additionally, it is a question of how to maintain these lists efficiently in a vectorized execution engine. In DuckDB, this would be feasible by using a ListVector, which was already discussed in Section 2.5, consisting of one flexible-sized vector for all the list elements and one `VECTOR_SIZE` vector providing the index per element of the vector for its corresponding chain. However, due to the next suggested emission strategy, this option was disregarded.

An improvement to the second alternative is to not materialize the hash table chains into a list but to append a pointer to the probe column instead, which is shown in Figure 4.16c. This pointer points to the head of the hash table chain. Upstream operators can access the tuples of the chain by following the pointers. In this case, it is not necessary anymore to copy the build side. Additionally, the cardinality of the join results remains lower or equal to the probe side cardinality. This would correspond with a d-representation in factorization theory [81], replacing the reoccurring subexpressions as the factor of a repeatedly probed hash table chain with its corresponding definition. This definition is the “physical” hash table chain, with different expressions (=probing rows) referring to it via the chain pointer. These definitions can lead to very efficient computing, as transformations like aggregations must only be computed once per definition, instead of for all expressions referring to it. How exactly we can operate on these will be introduced in the following section,

4.8 Factorized Execution

For the section on hand, we will focus on how we can use the d-representation discussed in the section above and depicted in Figure 4.16c to speed up the query execution time. While the 3D-Hash Join Paper [35] focuses on postponing the flattening of the result of a join after a selective second upstream join, we focus on computing aggregations on many-to-many joins and cyclic hash joins. In these cases, the flattening of the factorized intermediate can be avoided, which makes factorized processing very efficient.

4.8.1 Aggregates

When computing aggregates, there can be two cases. First, it could be that the aggregates do not involve the columns that are behind the hash table pointer and, therefore, in the definition of the factor. For example, for the query in Listing 4.4, the aggregation `COUNT(users.name)` $\cdots$ `GROUP BY Users.city` only groups by the `Users.city`. With the factor depicted in Figure 4.16c only consisting of the `source` columns, the aggregation does not need to access the elements in the factor. The second option is that the aggregates will be dependent on the factor’s columns.

For the first case, aggregates like `MIN(expr)`, `MAX(expr)`, `SUM(expr)`, `COUNT(expr)`, do not need to be expanded to compute the aggregate. For `MIN(expr)` and `MAX(expr)`, the factor can be disregarded completely. For the other aggregates `SUM(expr)` and `COUNT(expr)`, we still need to multiply the aggregate of the flat tuple with the length of its chain. For example, if we compute the count aggregate for the row

$$r_1 = \langle Eve\rangle \times \langle Bob\rangle \times \langle Berlin\rangle \times \overrightarrow{D}(\langle Alice\rangle \times \langle Bob\rangle) \tag{4.14}$$

with $\overrightarrow{D}(x)$ being the definition of x, we need to get the number of elements in the definition. This can be computed as $\texttt{COUNT}\overrightarrow{D}(x))$ which is 2. Therefore, $\texttt{COUNT}(r_1) = 2$ as there are two elements in $\overrightarrow{D}$.

To construct an example for `SUM(expr)`, we replace the 2nd column, which is Bob at the moment, with the number of habitats of Berlin. If we now want to compute the aggregate `SUM(users.habitats)` $\cdots$ `GROUP BY Users.city`, we have to multiply the aggregate of the row with the length of the definition. So for

$$r_1 = \langle Eve\rangle \times \langle 3,640,000\rangle \times \langle Berlin\rangle \times \overrightarrow{D}(\langle Alice\rangle \times \langle Bob\rangle) \tag{4.15}$$

we need to compute $3,640,000 \times \texttt{COUNT}(\overrightarrow{D}(x)) = 3,640,000 \times 2 = 7,280,000$.

So for the aggregates `SUM(expr)` and `COUNT(expr)` we still need the length of the definition. We can compute this length by first dereferencing the pointer of the definition, which might be a cache miss. Then we need to transverse the chain of the hash table, which can consist of hundreds of elements, each next pointer of the chain being another potential cache miss. So computing the length of the definition many times is a performance issue.

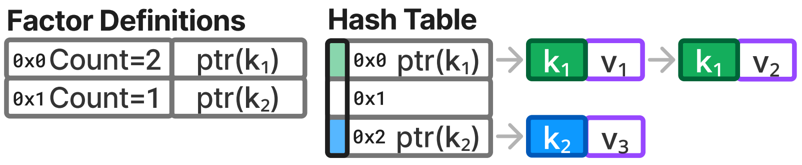


Figure 4.18: Cached `COUNT(expr)` aggregate per chain: Different probing tuples sharing this pointer can use the factor definitions' aggregate, avoiding traversal for counting.

We aim to solve this issue by instead of emitting pointers to the chains in the hash table directly, we emit pointers to so-called `factor definitions`. These themselves contain pointers to the actual chain head, but also space for aggregates of this factor like the count. Figure Figure 4.18 illustrates these factors. The array of factor definitions is allocated after the hash table build is completed. After this step, we know the number of unique chains in the hash table, so we know how many factor definitions we need. We then traverse the hash table buckets. If a bucket is empty, we simply skip it. If it is full, we save its pointer to a hash table chain to a factor definition and then the pointer to the factor definition in the bucket. Therefore, if this bucket later is probed, we will get a pointer to the factor definition.

If we then want to compute the aggregate of this chain for a probed row, we first dereference the pointer to retrieve the factor definition. We then check whether there is already an aggregate computed. If not, we traverse the chain using the chain pointer within the definition and count the number of elements. After this, we store the count in the cache field of the definition. If a second row now needs to compute the count, it only needs to dereference the definition pointer and retrieve the count from the cache.

While the hash table array size is the next power of two of the number of elements of the build side times two, the array of factor definitions is significantly smaller, only having the length of the number of unique keys in the build side. In our example of Figure 4.14 and Listing 4.4, the `follows` relation is built on the `dest` columns. As the following relation has 7 rows, the hash table array allocated by DuckDB is nextPowerOfTwo$(2 \times 7) = 16$. The number of unique elements of the `dest` column however is only 4. Therefore, the definitions array is four times smaller than the hash table array. This means, that there are much higher chances that the definitions array will fit into the CPU cache, making dereferencing pointers to their corresponding definition much faster than dereferencing pointers to the hash table array.

For some aggregates, it can also be that the aggregate is computed on columns within the chain. Similar to the count aggregate of the first case, we still can use the factor definitions to compute these aggregates. For example, if we want to compute the aggregate `string_agg(follows.source, ", ")` $\cdots$ `GROUP BY Users.city` to get a list of the followers per city. This aggregate now needs to retrieve the `follows.source` columns which are in the definition. For the example

$$r_1 = \langle Eve \rangle \times \langle Bob \rangle \times \langle Berlin \rangle \times \overrightarrow{D}(\langle Alice \rangle \times \langle Bob \rangle) \tag{4.16}$$

this would be the $string\_agg(\overrightarrow{D}(\langle Alice \rangle \times \langle Bob \rangle), ",") = "Alice, Bob"$. Again, instead of computing this aggregate over and over again, we can cache it in the factor definition.

To benchmark our factorized aggregate implementation[2], we created a micro-benchmark. We executed the benchmark on a MacBook Pro with an M1 chip using 1 thread, running the query five times and taking the median runtime. The query used is depicted in Listing 4.5.

[2] `https://github.com/gropaul/duckdb/tree/factorized-aggr`

```sql
SELECT o.ProductID, COUNT(p.PartID)
FROM OrderItems o
JOIN ProductParts p ON o.ProductID = p.ProductID
GROUP BY o.ProductID;
```

Listing 4.5: Query returning the number of parts needed per product for all orders

The `OrderItems` table contains 500,000 rows with 10,000 different `ProductIDs`. This results in an average of 50 duplicates per distinct `ProductID`. The `ProductParts` table contains 1,000,000 rows with 50,000 different `PartIDs` and 10,000 different `ProductIDs`. This results in an average of 100 duplicates per distinct `ProductID` and 20 duplicates per distinct `PartID`.

For the query in Listing 4.5, the `OrderItems` table is the build side as it is smaller. Since we build on `ProductIDs`, the hash table will have an average chain length of 50 elements. This means the intermediate result would be 50 times larger than the probing relation, assuming every key has a hit. By not expanding the result and instead returning chain pointers, the factorized intermediate result is only 2% the size of the expanded intermediate result.

We then probe for each `ProductID` on average 100 times. As we cache the length of one chain, we only need to traverse the whole chain 1% of the time. Therefore, we only need to calculate the length of the fact vector 1% of the time.

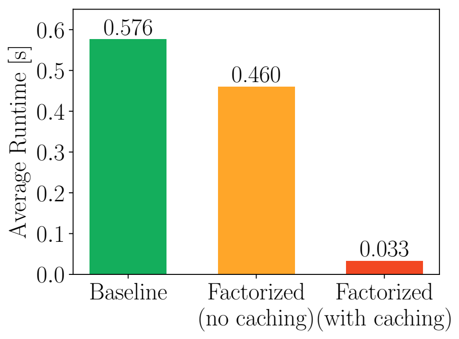


Figure 4.19: Runtimes for the microbenchmark in Listing 4.5 showing performance with and without caching elements in the factor definition

Configuration	**Average Runtime [s]**	**Speedup**
Baseline	0.576	-
Factorized (no caching)	0.460	1.25x faster
Factorized (with caching)	0.033	17.58x faster

Table 4.6: Runtimes and speedups for different configurations of the factorized aggregates microbenchmark

Figure 4.19 and Table 4.6 show the runtimes for the microbenchmark with three different configurations: baseline, factorized without caching, and factorized with caching. The baseline configuration has an average runtime of 0.576 seconds. When factorization is applied without caching, the runtime improves to 0.460 seconds, achieving a speedup of 1.25×. The most significant improvement is observed with factorization combined with caching, reducing the runtime dramatically to 0.033 seconds, resulting in a speedup of 17.58×.

From these results, we can conclude that only not expanding the intermediate result is not enough to get very sustainable performance increases. The problem is that if we only have the pointers referencing the chains, what we save on not having the intermediate result we have to give back by costly traversing the chains multiple times with costly cache misses. Only if we can avoid these traversals we get very good speedups.

While this micro-benchmark demonstrates the potential of factorization, such a query could also be computed with a group join [68], as both the aggregate and the join share the same key. This example is a simple illustration of computing aggregates in a factorized manner. However, many real-world scenarios could involve e.g. group-by clauses that depend on columns within the factor. This aspect is not within the scope of this thesis but presents an opportunity for future research.

4.8.2 Cyclic Joins

Cyclic joins provide another example where the intermediate result of the join operation is much larger than the final result. Figure 4.20 shows such a cyclic in a social media-like graph: the goal is to identify a pattern where a follows relationship forms a cycle, for example, among three users. In the case of a cycle involving three nodes, this is referred to as a triangle join.

```sql
SELECT *
FROM follows R, follows S, follows T
WHERE R.dst = S.src
  AND S.dst = T.src
  AND T.dst = R.src
```

Listing 4.6: Query returning all triangles within the follows relation

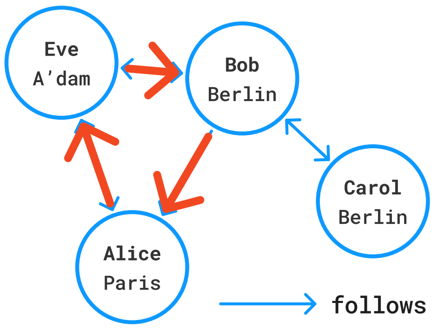


Figure 4.20: Example for a cycle in the context of social media

The SQL query in Listing 4.6 returns all triangles within the `follows` relation. It achieves this by joining the `follows` table three times, using aliases `R`, `S`, and `T`. Each alias represents a distinct instance of the `follows` relationship. The query identifies cycles (triangles) by ensuring that the destination of `R` matches the source of `S`, the destination of `S` matches the source of `T`, and finally the destination of `T` matches the source of `R`, closing the triangle.

The intermediate result is much larger because the condition `T.dst = R.src` closes the triangle. Before this condition, the intermediate result from the first join contained all candidates that had edges between the first two sides of the triangle. Only with the final selective join are the candidates reduced to those that include an edge between T and S, which are the edges that form triangles.

```sql
SELECT *
FROM R, S, T
WHERE R.a = S.a
  AND S.b = T.b
  AND T.a = R.b
```

Listing 4.7: Query returning all triangles within the relations R, S and T

As a running example to explain the binary join plan in detail and compare it against a factorized one, we chose the query in Listing 4.7. Note that here, the conditions are not

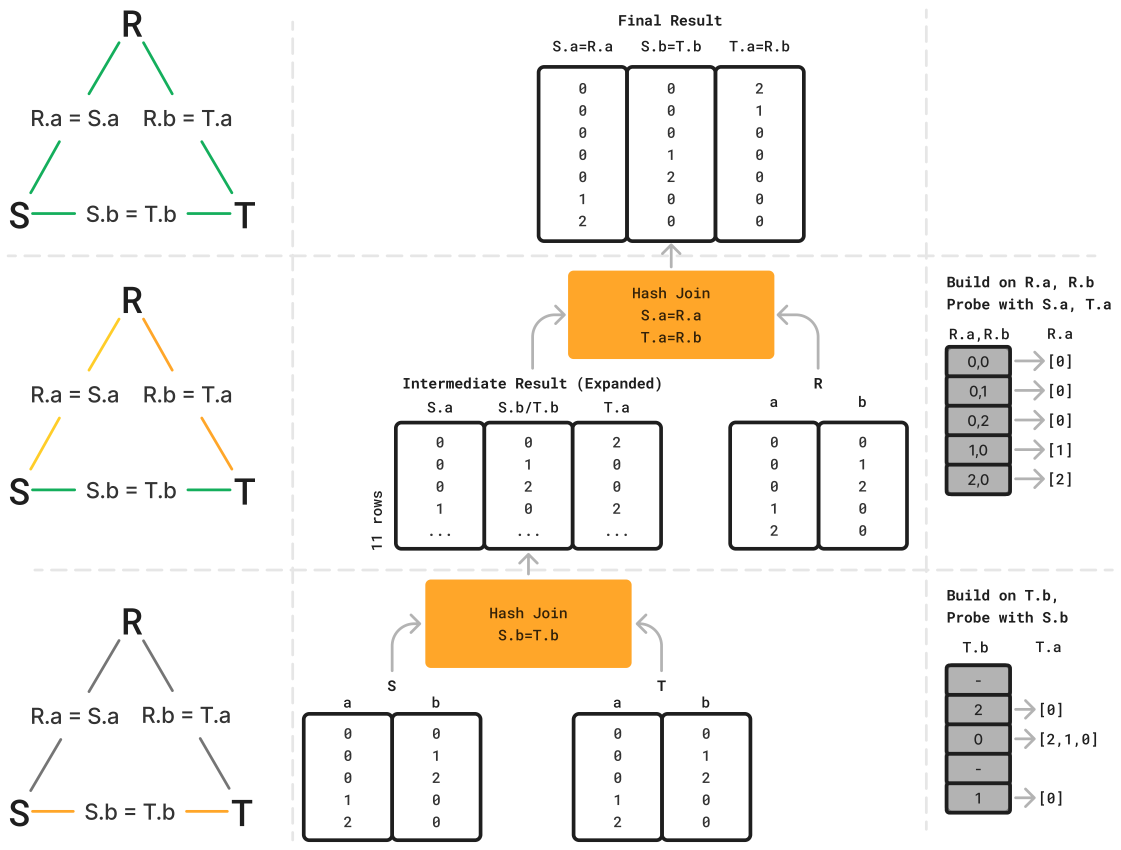


Figure 4.21: Binary query plan with intermediate results and hash tables for the query in Listing 4.7

"symmetrically", but still form a cycle. Instead of every condition following the pattern `X.a = Y.b`, the first two conditions have the pattern `X.a = Y.a`. In a graph context, this would mean that the required direction of the edge is changed in comparison to the first example.

In Figure 4.21 we can see the binary join plan for the query listed in Listing 4.7 together with its intermediate results. On the left-hand side of the figure, we can see the triangle together with its edge conditions. On the right-hand side, we can see the hash tables of the corresponding hash join operators. The query execution begins by retrieving all edge combinations in `S` and `T` that share a common node `b`. DuckDB uses a hash join for this, with `T` as the build side. Since we join on `S.b = T.b`, `T.b` is the build key for the hash table, and the chains of the hash table contain elements of `T.a`. Three rows qualify for `T.b = 0`, resulting in a chain of three elements for this hash table bucket. The other keys in `T` only occur once, so their respective chain only contains one element. This join operation is

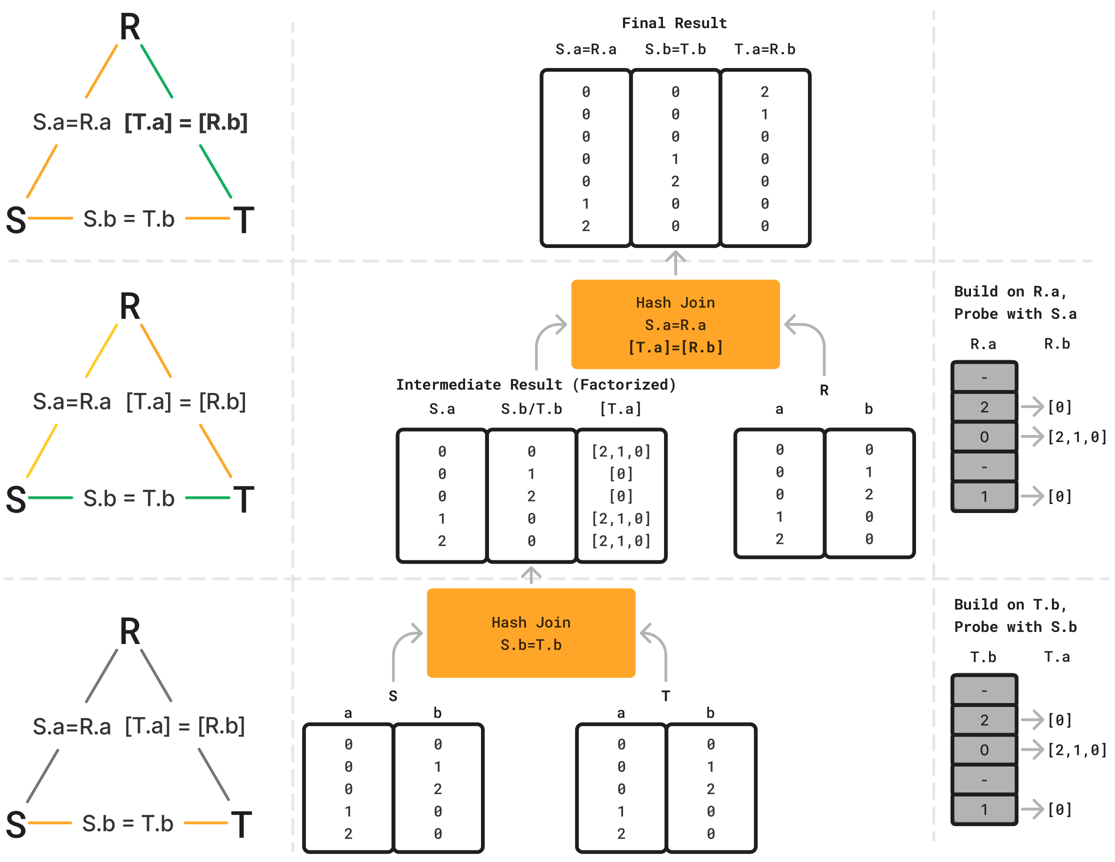


Figure 4.22: Factorized query plan with intermediate results and hash tables for the query in Listing 4.7

explosive, transforming the initial 5 rows into 11. The reason therefore is that every row in S for which `S.b = 0` holds will match with the three-element chain in the hash table, leading to three emitted rows per matching probing row.

This intermediate result becomes the probe side of the second hash join operator, with R as the build side. This join ensures the remaining triangle conditions are fulfilled by checking `R.a = S.a` and `R.b = T.a`. Consequently, the hash table for the second join is built on the composite keys `R.a` and `R.b`, ensuring unique entries in the hash table. The probing side computes the key (and hash) from `S.a` and `T.a`. This makes computing the last two join conditions very efficient, by checking both in one run. The final result, containing only seven rows, is smaller than the intermediate result of eleven rows.

In the factorized execution of the cyclic join, the goal is to avoid the explosion of the intermediate result of the first join and to efficiently find matches for the factorized tuples. The whole process is depicted in Figure 4.22. Both plans start with the same hash join

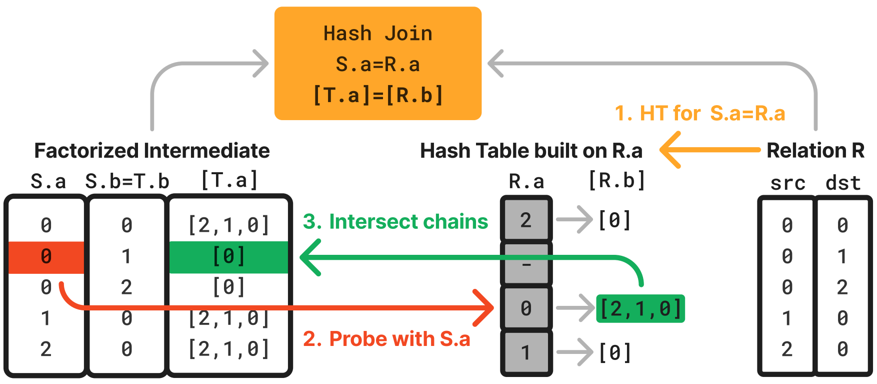


Figure 4.23: Detailed illustration of the second join probe and factorized condition checking for the factorized join plan in Figure 4.22

operation combining `S` on the probe side and `T` on the build side under the condition `S.b = T.b`. Again, the operator builds a hash table on `T.b` which contains chains of elements of `T.a`. But instead of the explosive flat result, the operator emits the pointers to chains of `T.a`. Therefore, we only have five rows in the intermediate results. Note that in Figure 4.22, the intermediate result now has a column of `[T.a]` instead of `T.a` compared to the binary join plan. In addition, the column's values are depicted as actual lists, but physically they only contain pointers to these lists.

In the cyclic join, the second hash join needs to check the factorized condition `[T.a] = [R.b]`. Therefore, we need chains containing elements of `R.b`. This is achieved by only building the hash table for the second operator on the key `R.b`. The resulting hash table is depicted on the right center of Figure 4.22.

In the probing phase both the flat condition `S.a = R.a` and the factorized condition must be checked. This happens in three steps, which are depicted in Figure 4.23. The graphic shows only the second join of Figure 4.22 without the result of the join. The first step is to build the hash table for the flat condition `S.a = R.a`, which than has chains containing values of `R.b`. For each element with have to probe, we first satisfy the first condition by probing the hash table with the key `S.a` of the probe side. This is depicted as a red arrow in Figure 4.23. After finding the corresponding bucket in the hash table, we know that all elements in the chain of this bucket hold true for the condition `S.a = R.a`. To check the factorized condition, we have to intersect this hash table chain `[R.b]` now with the probe side chains `[T.a]`, which can be accessed using the probe side pointers. This is depicted as the green arrow in Figure 4.23. The figure shows this process for the

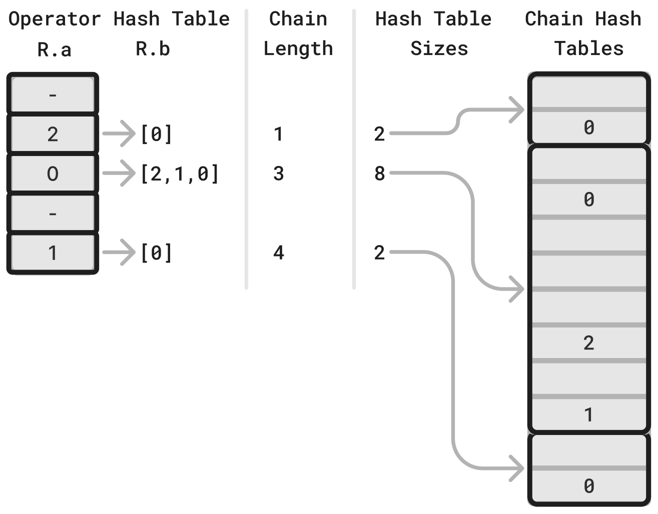


Figure 4.24: Allocation of micro hash tables for evaluating factorized join predicates

second row r_2 of the probe side:

$$\texttt{[T.a]}_{r_2} \cap \texttt{[R.b]}_{S.a=R.a=0} = [0] \cap [2,1,0] = [0] \tag{4.17}$$

Note that the intersection operation $\cap$ is not strictly a mathematical set operation. For every element in the probe side chain C_p, we need to iterate over each element of the build side chain C_b and return a match if they satisfy the joining condition. Therefore, this intersection is more like a conditional Cartesian product - $C_p \bowtie_{cond} C_b$. For example, if $C_p = [1,2,2,3]$ and $C_b = [2,2,1]$, the result would be $[2,2,2,2,1]$, since every $2 \in C_p$ must be paired with every matching $2 \in C_b$.

To compute the entire join, we perform many "micro" joins or list intersections between the probe side factor chain and the matching build side check. Although these lists are typically short, these micro joins might need to be computed millions of times, once for each row on the probe side, making efficient implementation critical.

Similar to the "big" joins, the micro joins are also computed using hash tables. Therefore, for each of the chain intersections, we need to choose one chain as the probe and one as the build side. Then, we build a tiny hash table for the build side chain and probe this table with the elements of the probe side chain. This process and the population of the hash tables is depicted in Figure 4.24: For each of the chains in the hash table, we calculate the size of the hash table as the next power of two of two times the number of elements in the chains. We use a power of two as the size to be able to replace the modulo to range the hash to the size of the hash table array using a bitmask, similar to the standard hash table of the hash join operator. Details can be found in Section 4.4.2.

The information on how big the hash table for a chain is and where it is located in memory is again stored in the factor definitions which were previously used to cache the count in the factorized aggregate calculation (see Figure 4.18). When processing cyclic joins, these definitions store the pointer to the chain's hash table as well as additional information like whether this hash table was already built or not.

Since we do not know if a chain will be chosen as the build side of a chain intersection, we do not populate the allocated arrays directly. Each chain, however, already has an array it could use for building the hash table. During the probing phase, if we determine that two chains need to be intersected, the build side is chosen as follows: If both or neither of the chains already have a build hash table, the *larger* side is chosen as the build side. If one of the chains already has a build hash table, it is chosen as the build side. We chose the larger side as the hash table since building the hash table is a reusable task for many probes. Building the hash table for a large chain and probing it multiple times with many shorter chains is more efficient than repeatedly probing with the large chain on many small hash tables. Further work might also evaluate the performance of always taking the larger relation as the build side to truly achieve the worst-case optimal join guarantees, but this has not been implemented in this project.

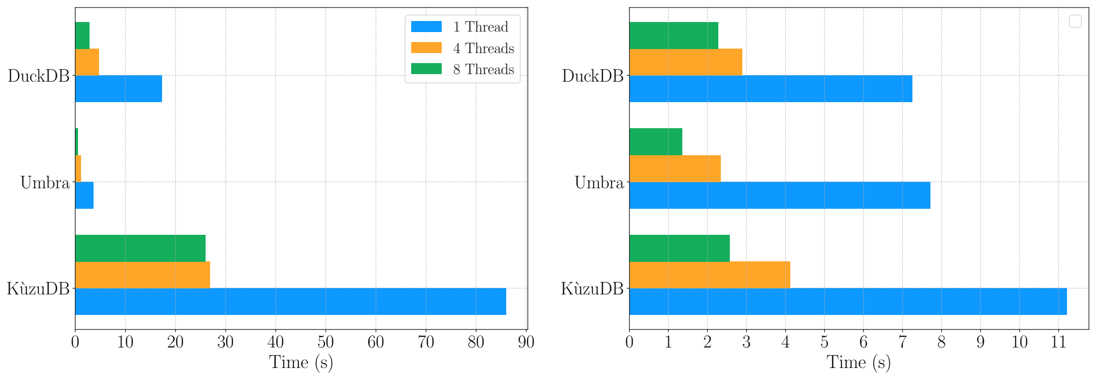
After two of the hash table chains are interested, we emit a flat result as the join operations result as we cannot use the chains anymore because each probing tuple now has the unique intersection between probing and build side chain. Instead of emitting a flat result, one could also consider emitting a factorized result with a materialized chain, similar to the one depicted in Figure 4.16b. Investigations into this could be interesting for future work.

```sql
SELECT COUNT(*) FROM links l1
JOIN links l2 ON l1.dst = l2.src
JOIN links l3 ON l2.src = l3.dst AND l3.dst = l1.src;
```

Listing 4.8: Count the number of cycles in the WebStan dataset

We evaluated the performance of factorized cyclic joins in DuckDB by performing triangle joins along links in the WebStan dataset[3]. This dataset consists of 685,231 nodes and 7,600,595 edges. The cyclic join experiment was conducted on a cloud instance running Fedora 40, equipped with an Intel Xeon Gold 5115 CPU with 40 threads and 248 GB of RAM. The query used to find triangles among the hyperlinks is shown in Listing 4.8. We

[3]`https://snap.stanford.edu/data/web-BerkStan.html`



(a) Binary Join Plan **(b)** WCOJ/Factorized Join Plan

Figure 4.25: Runtimes of different systems and configurations for finding the number of triangles in WebStan dataset

compared the binary join plan execution time with the WCOJ-plan for the systems Kùzu [34], Umbra [76], and DuckDB [84], which all have implementations for both algorithms. To test the scalability of the systems, we ran the query with one, four, and eight threads per system.

The results of these benchmarks are depicted in Figure 4.25 and detailed in Table 4.7. Furthermore, the parallel efficiency of the algorithms is depicted in Table 4.8. Parallel efficiency is a measure of how effectively a parallel algorithm utilizes multiple processors. It is calculated as the ratio of the speedup achieved with multiple processors to the number of processors used. High parallel efficiency indicates that the algorithm scales well with the addition of more processors. The formula for parallel efficiency E_p is given by:

$$E_p = \frac{T_1}{T_p \times p} \tag{4.18}$$

where T_1 is the execution time with 1 thread, T_p is the execution time with p threads, and p is the number of threads.

For the Binary Join plan, Umbra outperformed both Kùzu and DuckDB across all thread counts, outperforming DuckDB by a factor of four. Kùzu's binary join plan is the slowest, being outperformed by DuckDB by a factor of about four for one thread. In terms of scaling with CPU cores, DuckDB exhibits the best scaling for four cores with a parallel efficiency of 90.9%, while for eight threads, Umbra achieves the best scaling with an efficiency of 77.0%.

For the worst-case optimal join plans, or in the case of DuckDB factorized join plans, the runtime numbers are closer together. For one thread, the factorized execution strategy

Table 4.7: Runtimes of different systems and configurations for finding the number of triangles in WebStan dataset. While for the binary join plan, Umbra is superior, for the factorized join plan and one thread the new DuckDB factorized join plan is the best

Algorithm	System	1 Thread (s)	4 Threads (s)	8 Threads (s)
Binary Join	Kùzu	86.010	26.913	25.973
	Umbra	**3.645**	**1.158**	**0.585**
	DuckDB	17.337	4.767	2.867
WCOJ	Kùzu [34]	11.214	4.126	2.572
	Umbra [76]	7.715	**2.340**	**1.356**
	DuckDB [84]	**7.253**	2.889	2.272

Table 4.8: Parallel Efficiency of the Systems in table Table 4.7 for four and eight threads.

Algorithm	System	4 Threads	8 Threads
Binary Join	Kùzu	0.799	0.414
	Umbra	0.787	**0.770**
	DuckDB	**0.909**	0.756
WCOJ	Kùzu	0.679	0.545
	Umbra	**0.824**	**0.711**
	DuckDB	0.628	0.399

proposed in this thesis outperforms Umbra by 0.5 seconds, being the fastest implementation. However, for four and eight threads, Umbra's multijoin implementation is faster. The Kùzu WCOJ implementation has the largest execution time but comes close to DuckDB for eight threads. As shown in Table 4.8, both DuckDB and Kùzu have parallel efficiencies of around 65% for four threads, while Umbra excels with 82.4%. The DuckDB factorized implementation has a comparatively low efficiency for eight threads at only 39.9%. This is due to the lack of optimization in parallelization, with steps like the micro hash table memory allocation occurring sequentially.

Table 4.9: Speedup of WCOJ over Binary Join for Different Systems and Threads

System	1 Thread	4 Threads	8 Threads
DuckDB	2.390	1.650	1.262
Kùzu	7.670	6.523	10.098
Umbra	0.472	0.495	0.431

To determine which systems benefit from factorized or multi-join execution, we can refer to Table 4.9. It illustrates the speedup of the WCOJ implementation over the binary join plan. Notably, Kùzu achieves the highest speedup compared to their binary join implementation, with a speedup of 7.7x for one thread and 10.1x for eight threads. DuckDB shows a significant speedup with smaller thread counts, achieving 2.4x for one thread but only 1.3x for eight threads, showing that factorized execution is not yet optimized for parallelism. Umbra, however, does not benefit from WCOJ optimal joins; their binary execution is more than twice as fast as the WCOJ plan, with a speedup of 0.5x.

5

Adaptive Factorization

As shown in the chapter before, factorized execution, or WCOJ, can outperform binary join plans by avoiding the creation of large intermediate results. However, this approach requires additional helper structures and more complex logic to compute the results. For instance, when computing cyclic joins using a factorized execution strategy, multiple small hash tables are needed to perform list intersections. These investments may not justify the runtime benefits if large intermediate results are not avoided. In the case of cyclic joins, if there are few duplicates on the build side keys, the chains of hash tables used as factors may be short or, in the worst case, contain only one item.

This is why, while WCOJ and factorized query processing offer great alternatives, they cannot fully replace binary join operators and require planning by the query optimizer. The challenge of determining the optimal conditions for using these algorithms remains unresolved, as discussed in Section 3.3.

In this chapter, we aim to assess a new approach for this challenge by introducing an adaptive approach that decides on runtime when to apply factorization. The idea is that if we identify a query that could benefit from factorization like a cyclic join or a many-to-many join followed by an aggregation during the query optimization phase, we mark the plan as potentially factorizable. This does not change the execution strategy directly but triggers the collection of runtime metrics. We delay the decision of whether to use factorization until the factorized and default query execution start to differ. Based on the runtime metrics we have gathered, we now have a much clearer understanding of the data structure, allowing us to assess the potential benefits of factorization more precisely.

The work presented in this chapter was created in close collaboration with Daniël ten Wolde.

5.1 Adaptivity through Run-time Strategy Switching

The core concept is to adaptively implement factorization by postponing the decision to use either a factorized or flat execution strategy until the latest possible moment and collecting metrics until this moment.

Our proposed approach for integrating factorization and, worst-case, optimal cyclic join computation is based on altering and repurposing existing data structures without introducing completely new operators and concepts. Therefore, traditional query plans do not usually differ drastically from plans employing these new technologies. This was also illustrated in the flat and factorized query plan for finding triangles in Figure 4.21 and Figure 4.22. Here, binary and worst-case optimal execution are similar in the beginning. The first join is nearly the same in both plans, only differing in the probing phase. This shows that it is advantageous to avoid completely different operators, such as WCOJ or multi-join operators. Instead, we can make small modifications to the Hash Join Operator to switch from a standard to a factorized execution strategy.

For the second join, during materialization, for both execution strategies, the hash of both build side keys is needed. In a binary join plan, the hash table is built on the composite of the two keys, so we need a hash that includes both. For a factorized execution, a hash for the first key is needed to build the hash table, while keys for the second join are used to perform micro hash joins later. Therefore, the hashes of both keys are required in both scenarios. These hashes can, therefore, be used to compute sketches, which provide additional metrics for the runtime decision.

The decision on when to opt for factorization can be made after the first hash table is built but must be made before probing the first hash table. Additionally, it can occur after the materialization phase of the second hash table but before the build of this hash table. As we can wait until the first join hash table is built, we get an accurate distinct count for the build key of the first hash table as a metric. This count enables us to determine the average chain length, calculated as the distinct count divided by the number of elements on the build side.

This shows that an adaptive hash join operator can postpone the decision on whether to use a factorized or normal execution strategy very late in the query execution process. At this point, we will have better statistics on our data than during the query optimization phase. Instead of relying on cardinality and unique value estimations, we can (additionally) use accurate metrics. Which metrics we collect will be discussed in the next section.

5.2 Training and Evaluation Dataset

To gather the required metrics for training and evaluating the adaptive factorization and our classifiers, we used three distinct datasets. In each dataset, we constructed triangle queries that, while searching different relations for triangles, all represent cyclic queries.

The first dataset was a synthetic collection of 305 graphs, with sizes ranging from 100 to 100,000 nodes and 100 to 10 million edges. These graphs were designed with a broad range and distribution of in-degrees and out-degrees, and we generated them using the Python library NetworkX [75].

The second dataset included SNAP datasets [60], where we took graphs containing over 1 million edges. These datasets provided real-world data for our analysis.

Finally, we used the LDBC SNB BI datasets [94], using scale factors of 1, 3, and 10. From the SNB dataset, we extracted 352 unique triangle queries on different relations. To get to this number of triangle queries, we additionally created reverse versions of each of the 27 edge tables by swapping the source and destination columns and using them to find more triangle queries. We combined all these datasets into one big data set.

As the effort of implementing an adaptive query execution in DuckDB is large, to evaluate the viability of our approach we proceeded as follows. For each of the queries, we run both the factorized execution strategy and the flat execution strategy. We measure the median execution time of five runs for each strategy. While running the factorized strategy, we also logged the runtime metrics so we could use them for later analysis. As we have the runtime for both strategies per query, we can compute the speedup of the factorized strategy and use this as the target column for the models. We then can try to use the collected runtime metrics to train and test a classifier to predict this speedup based on the metrics.

In Table 5.1, we present a sample of the collected data, showing the first four rows. Each row represents a single query and includes various metrics collected during execution, such as join sizes, chain lengths, predicted join size, the number of unique relations, and runtime metrics. These metrics, their retrieval, and their significance will be in detail discussed in the next section. The speedup column indicates how much faster the factorized strategy was compared to the flat strategy, and the ”Faster Class” column marks whether the factorized strategy was faster (1) or not (0). The final collected data consisted of 3998 rows.

Columns	**Row 1**	**Row 2**	**Row 3**	**Row 4**
1st Join Size	1,801,048	596,297	1,600	205,858
1st Join Avg Chain Length	18.01	596.3	16	205.86
1st Join AMS Array Mean	0.0008	0.01	0.0375	0.0101
2nd Join Size	1,801,048	499,858	15,040	205,858
2nd Join Avg Chain Length	18.01	499.86	1.18	205.86
2nd Join AMS Array Mean	0.0008	0.01	0.0383	0.01
Predicted Join Size	35,098,596	336,036,013	242,704	47,885,800
Number of Unique Relations	1	3	3	1
Explosion Factor	19.49	613.12	29.17	232.62
Baseline Median Runtime	1.65	9.98	0.014	1.33
Factorized Median Runtime	0.96	7.34	0.022	0.79
Speedup	1.72	1.36	0.63	1.68
Faster Class	1	1	0	1

Table 5.1: Sample Data from the Collected Dataset used to train and evaluate the adaptive factorization

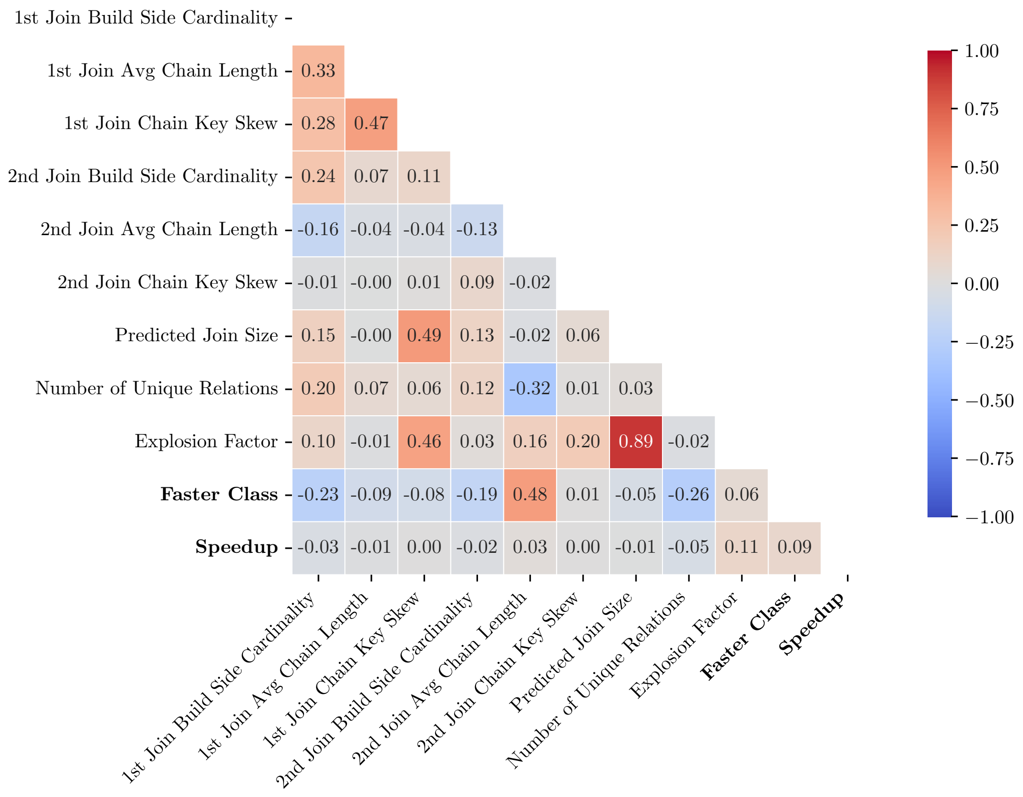


Figure 5.1: Correlation Matrix of the Metrics:

5.3 Metrics for Adaptive Factorization

In order to make a sound runtime decision, we consider both runtime and optimization time known metrics. Our experiments show that collecting the runtime features only leads to a runtime overhead of below 1%.

Figure 5.2 presents a density diagram illustrating the distribution of speedup across different metrics. The plots suggest a correlation between certain metrics and the observed speedup. What becomes apparent is that the speedup for the factorized triangle joins heavily varies per query, as the dataset features speedups in the range of 0.1 to 10. This again highlights how important it is to find a good method to decide whether to use the factorized or the normal execution strategy.

To assess the potential explosiveness of the join result, we estimate the join size of the condition that would be computed using list intersection. For example, in Figure 4.22, this would correspond to the condition `[T.a] = [R.b]` in the second join. We use the AMS sketch, a type of linear sketch, to estimate the join size between relations, which enables the detection of skew [6].

During the hash table build phase, an AMS sketch is constructed for the keys that will be compared using list intersection, `T.a` and `R.b`. The sketches generated from both hash tables can then be multiplied to estimate the size of the intersection join.

The AMS sketch is represented by a matrix C with dimensions $d \times w$, where d is the number of hash functions used, and w is the number of buckets for each hash function. Two hash functions are employed: (1) $h_j(i)$ maps item i to a bucket in the j-th row, and (2) $g_j(i)$ maps item i to either $+1$ or -1, adding randomness and enhancing the independence of updates.

In our implementation, we utilize the 64-bit hash generated during the hash join materialization phase for each item i. Since this hash must be computed in both the binary and factorized join plans, there is no additional overhead in hashing the keys for the sketch. This 64-bit hash is split into bytes, and for each row j, the j-th byte is used for both $h_j(i)$ and $g_j(i)$. Specifically, $h_j(i)$ is determined by the last 7 significant bits, while $g_j(i)$ is determined by the most significant bit, mapping to either $+1$ or -1. We use the AMS sketch to estimate the explosiveness of the join keys by populating two sketches with the hash values of the respective keys and taking their scalar product as an estimate of the join size.

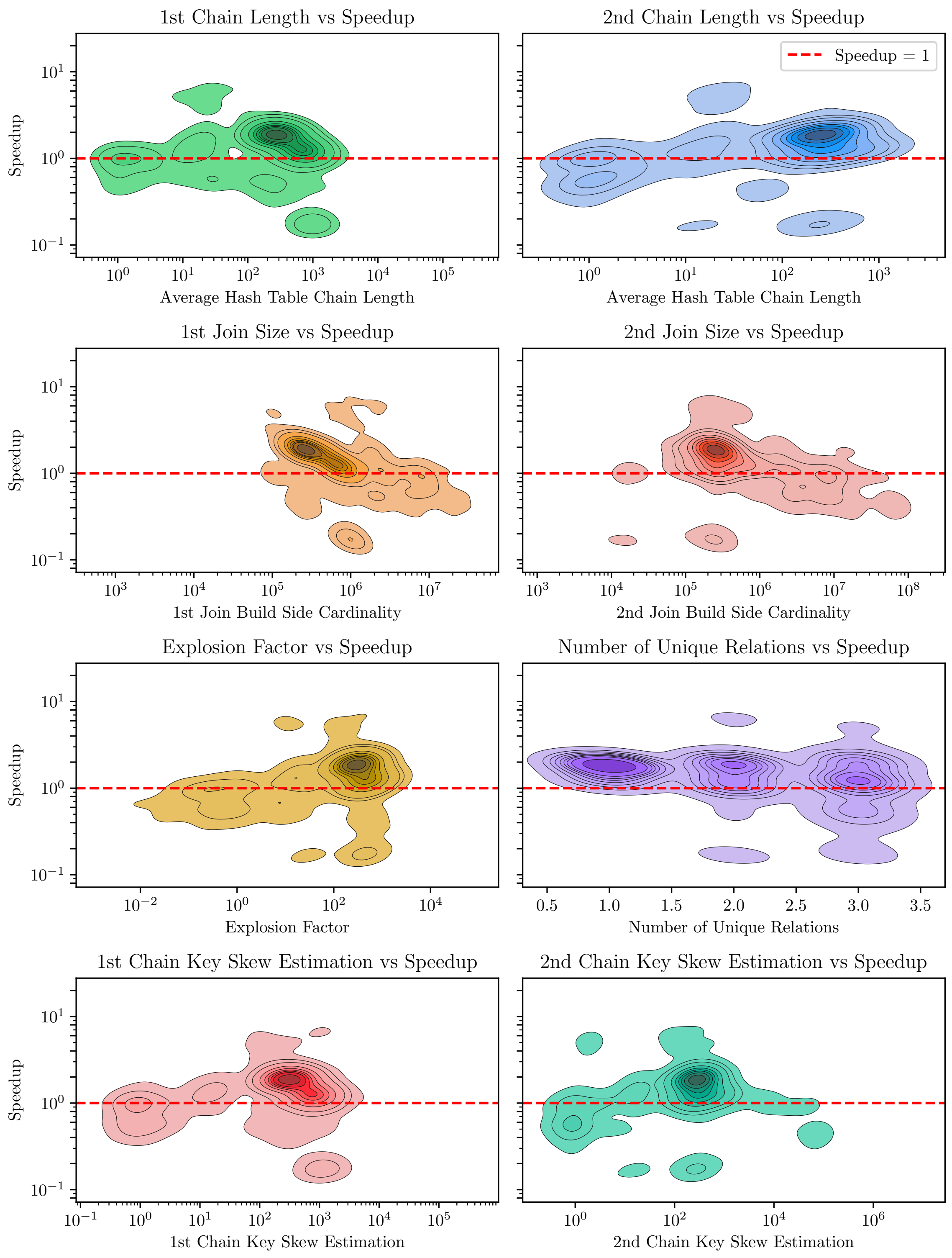


Figure 5.2: Distribution of the Speedup over different static and runtime metrics:

To further understand the relationships between the metrics and their influence on the speedup of factorized queries, we compute a correlation matrix visible in Figure 5.1. This matrix shows how strongly each metric is linked to the observed speedup. The correlation coefficient, ranging from -1 to 1, indicates the strength and direction of the relationship: positive values show that as one metric increases, so does the speedup, while negative values indicate the opposite. The matrix is computed using the Pearson correlation coefficient, which measures the linear correlation between two variables [82]. The matrix displays the features used for prediction, along with the metrics to be predicted, which are the faster class and the speedup. Using this matrix, we will will now discuss the features' quality.

1st Join Average Chain Length

Retrieval This feature describes the average length of the chains of the hash table that is built on the first hash join operator. In Figure 4.22, this is the join having the condition `S.b. = T.b`. As discussed, this join operator only needs to either emit factorized or flat tuples during the probing process depending on the execution strategy. Therefore we can collect metrics while the hash table is built, as we only have to decide on a strategy after the build is completed. We can get the average chain length by having a counter that gets incremented every time we insert an element into an empty slot. Thereby we get the number of chains we have in the hash table. In addition, we know the number of elements of the build side after the sink phase of the hash table is completed (see Section 4.4.1). To get the average number of elements in the chain, we divide the number of elements of the build side by the number of chains.

Quality The distribution of the average chain length over the speedup is depicted in the top left corner of Figure 5.2. We can see there is some correlation with the speedup in the range of a chain length of 1 to 100, with the speedup getting higher the longer the chain is. However, in between a chain length of 100 to 1000, the speedup seems to decrease with a larger chain length. Looking at the correlation matrix, we can see that there is only very little linear correlation between the first chain's average length and both faster class and speedup. However, it could be that there is still some correlation as seen in the distribution plot, but just not a linear correlation.

2nd Join Average Chain Length

Retrieval This feature, similar to the first, describes the average chain length of the hash table built during the second hash join. However, unlike for the first feature, we cannot wait for the hash table to be fully constructed, as the final hash table will vary depending on the chosen execution strategy. Despite this, the materialization phase of

the join in the two strategies remains the same, allowing us to populate an HyperLogLog (HLL) sketch [36] using the hashes of the keys computed during this phase. This sketch can be used for estimating the number of distinct values. With this, we can again calculate the average chain length, as these unique values represent the number of chains. For the project on hand, we used the exact distinct values that are available after the second hash table is built during the factorized execution strategy, as there was not enough time to implement the HLL sketch. Using an estimate instead will be part of the future work.
Quality The distribution plot of this feature can be found in the top right corner of Figure 5.2. This feature seems to be very promising, as there seems to be a linear correlation between the speedup and the chain length: The longer the chains of the second join, the higher the speedup. This is also suggested by the correlation matrix, as the correlation between the faster class and this feature is the best compared to all the other features.

1st & 2nd Join Build Side Cardinality
Retrieval These two features can be discussed together as both the retrieval and the quality are very similar. They describe the cardinality of the build side for the first or second join, respectively. The exact cardinality of the build side is known after the sink phase of the hash join operator is completed, as the operator waits for all the tuples to materialize to determine the hash table size.

Quality Both the first and second build side cardinalities appear to be negatively correlated with speedup: larger build sides tend to result in lower speedups. This is also visible from the correlation matrix, where both cardinalities have a negative correlation of around -0.2. Thus, these metrics are valuable as features for a prediction model. This correlation appears counterintuitive, as factorization's potential should not depend on data cardinality. Therefore, the bad performance of the factorized approach for large data could be because of the lack of optimization for parallel processing. In general, with a growing data size, the parallel efficiency should increase. However, since the factorized approach offers less parallelization than binary methods, larger datasets might more clearly expose this limitation.

Predicated Join Size & Explosion Factor
Retrieval The predicated join size comes from the join size estimation using the AMS sketches described in the text above. However, as this metric is very dependent on the size of the join, we decided to introduce a second metric, the explosion factor. This metric is

calculated by dividing the predicted join size by the average of the cardinalities of the two build sides.

Quality The explosion factor's distribution over the speedup is depicted in the 2nd last row and first column of Figure 5.2. We can see that a high explosion factor seems to have an influence on the speedup: the bigger the explosion factor, the higher the speedup. Looking at the correlation matrix, we can only see a very small correlation between the labels and the explosion factor or the predicted join size.

Number of Unique Relations

Retrieval The number of unique relations across which triangles are searched can be determined from the query plan, making it a static metric that is known before runtime. If there is only one unique relation, cycles are searched within that relation, for example, cycles within the `follows` relation. If there are two unique relations, the query involves different relationships, like finding users where one follows another, and both live in the same town, involving the `follows` and `lives_in` relations. This metric is interesting because, with only one relation, metrics like the first hash table chain length could also be relevant to the second hash table chain length when joined on the same key. Additionally, if the probe side joins keys match any of the build side keys, we can infer information about them as well.

Quality The distribution plot for this metric is shown in the second column of the third row in Figure 5.2. The number of distinct relations has a noticeable impact on the speedup distribution. With more relations involved, the range of possible speedups becomes wider. When only one relation is involved, nearly all queries achieve a speedup greater than 1. However, with three relations, the speedup distribution is much broader, and more queries tend to have speedups closer to 1.

1st & 2nd Chain Key Skew Estimation

Retrieval To get information in addition to using the AMS sketch to predict the join size we also use it to get information on the skewness of the join keys involved in the factorized intersection. This is done by instead of calculating the F_2 value on the two sketches of both sketches, we calculate it based on twice the same sketch. This returns the size of a self-join, which can then be used to estimate the number of duplicates in the sketch.

Quality The distribution over the speedup for both estimations can be seen in the last row of Figure 5.2. This feature seems to be loosely correlated with the speedup, getting

Selection Method	**Accuracy**	**Speedup**
Objective Best/Worst		
Always choose faster	N/A	1.58x
Always choose slower	N/A	0.90x
Always choose factorization	N/A	1.47x
Never choose factorization	N/A	1.00x
Classifiers		
Logistic Regression	0.78	1.47x
Decision Tree	0.87	1.54x
Random Forest	**0.89**	**1.55x**
Gradient boosting	0.88	1.55x

Table 5.2: Accuracy and Speedup for various selection methods and classifiers on the test dataset

higher speedups the larger the skew estimation. However, the correlation matrix does not show a big correlation.

5.4 Adaptive Factorization with Machine Learning Strategies

Using the features outlined in the previous section, we trained machine learning models to predict whether it would be advantageous to switch to factorized execution for a given query based on its characteristics. We used an 80:20 train-test split to train and evaluate the performance of our models.

Table 5.2 compares the accuracy and speedup of different selection methods and classifiers on the test dataset. The "Always choose faster" scenario represents the best possible situation with a perfect model, where the fastest execution strategy is always selected. It achieves the highest possible speedup of 1.58x, setting the upper bound for what any predictive model could achieve. In contrast, the "Always choose slower" scenario assumes the slowest execution strategy is always chosen, leading to a speedup of 0.90x, which serves as the worst-case scenario and lower bound for performance. "Never choose factorization" is the baseline, consistently applying the standard execution strategy. The "Always choose factorization" means always selecting the factorized execution strategy, resulting in a speedup of 1.47x. This shows that factorization is generally faster than the baseline but still slower than the perfect strategy. Therefore, the range between the factorization speedup (1.47x) and the perfect model (1.58x) represents the potential performance gain our machine learning models can achieve.

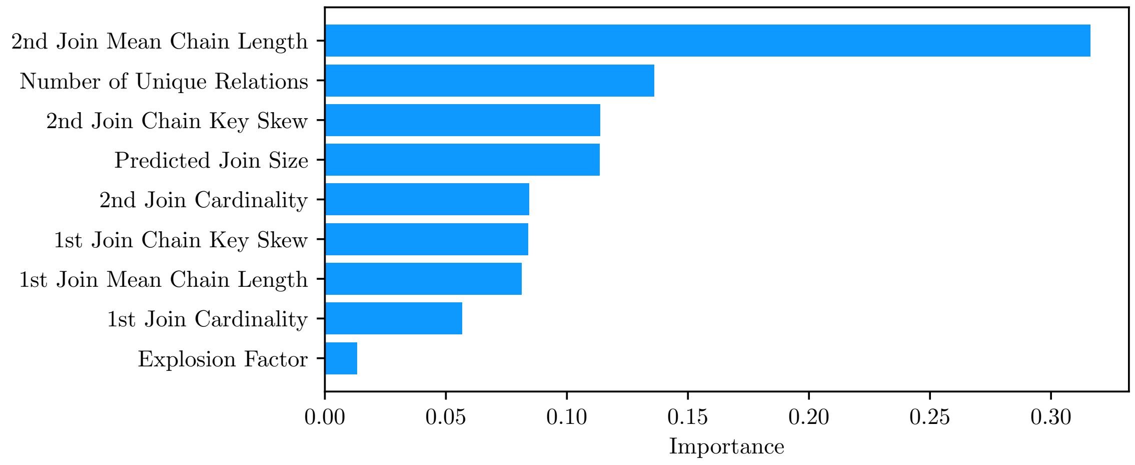


Figure 5.3: Features and their importance for the random forest model indicate that the 2nd join's mean chain length is the most important, with both dynamic (chain length) and static (number of unique relations) features being important.

The best strategy was achieved using a Random Forest classifier, having an accuracy of 0.89 and a speedup of 1.55x, which is nearly optimal compared to the maximum possible speedup of 1.58x. Other models, including Decision Trees and Gradient Boosting, also showed good performance, with only slight differences in accuracy and speedup.

Figure 5.3 shows the importance of the features used in the Random Forest model. The 2nd join's mean chain length is the most important feature. This was to be expected from the analysis of the metrics in the previous chapter. It is twice as important as the following feature. These are the number of unique relations, which is a static feature, and the 2nd join key skew, as well as the predicted join size. Therefore, we can conclude that the runtime metrics greatly improve the model's performance and that using these features, machine learning models can predict accurately when the factorized execution strategy would be faster than the default execution strategy.

6

Discussion & Future Work

We showed that by introducing our Linear-Chained Hash Table to the Hash Join Operator in DuckDB to have collision-free chains, we cannot only increase the performance for large data sets and many-to-many joins but also use these chains as d-representations for factorized query processing.

The alterations made to the hash table are well-tested and production-ready and were merged into DuckDB[1]. They are expected to become publicly available with DuckDB 1.1, which is planned to be released on the 2. September of 2024 [100].

One possible further optimization for the hash table could be to not only use the free upper bits of the hash table pointers as salt. As we have 16 free bits, there is only a chance of $\frac{1}{2^{16}} = \frac{1}{65{,}536}$ that if the salt of two values matches, the keys do not match. In this case, the key comparison was unnecessary. We could repurpose one of the 16 free bits as a marker which indicates whether there was a hash collision for a slot during the build of the hash table. If we now probe for a key and find that the hash table slot, which the hash of the probing key leads to, is full but does not match the probing key, we can use the marker. If the marker indicates that there was no hash collision during the build, we know that linear probing is unnecessary and that there is no need to check the next slot. Otherwise, we have to check the next slot. Re-purposing this bit increases the chance for a false salt match to $\frac{1}{32{,}768}$, which should still be enough. Implementing and benchmarking this optimization would be interesting for further work.

While the changes to DuckDB's Hash Table layout lay the foundation for making factorized query execution publicly available in DuckDB, the factorized strategies discussed in this work have only been implemented as a proof of concept. This includes the

[1] `https://github.com/duckdb/duckdb/pull/11472`

implementation for factorized aggregate computation only for the `COUNT(*)` aggregation[2]. However, we could demonstrate that, in this case, factorized query processing can yield significant performance improvements. Furthermore, we could show that integrating factorized query processing does not require new operators but that it is enough to update existing operators to handle factorized vectors. For aggregations, further work should include supporting all types and all aggregates DuckDB supports to fully integrate factorized aggregate computation into DuckDB. In this work, we only introduced aggregates for the case where the `GROUP BY` key is not within the factor itself. It would be interesting to see how one could efficiently implement factorized aggregations for the case where the key is within the factor. This could allow factorized aggregates for all cases where there is a build side with duplicate keys followed by an aggregate function.

One question that remains open in regards to factorized aggregation is how the approach proposed by us relates to the claim of Birler et al., stating that computing aggregates on factorized representation is not necessary because of eager aggregation [19]. Further work should analyze which cases can adopt factorized aggregation or eager aggregation and how the two approaches are related in terms of performance.

Also, for the cyclic join, this work only features a proof of concept implementation[3]. The implementation can compute most n-sized cyclic queries, with triangle queries being the most tested. The hash table used to intersect the chains is currently a custom linear probing hash table but should be altered to use the same infrastructure as DuckDB's Hash Table for joins. This would allow us to support all different types of join keys, while the current implementation is only usable with 64-bit integer keys.

Furthermore, as seen in the evaluation of the cyclic join, the factorized execution strategy currently lacks optimization for parallel execution, with a parallel efficiency of only 39.9% for eight threads. While DuckDB can beat Umbra's WCOJ implementation in single-threaded mode, using eight threads, it is nearly two times slower than Umbra. This is why the parts of the algorithm that currently happen sequentially must be parallelized. Here, the focus should mainly be on the allocation and distribution of memory for the micro hash tables and the creation of the factor definitions described in Figure 4.18.

Another method of using factorization that we did not explore in this work is delaying the expansion of one join operator's chains after a second, selective join, which was proposed by the 3D hash join paper [35]. To implement this, we would need an expansion operator that

[2]`https://github.com/gropaul/duckdb/commits/factorized-aggr/`

[3]`https://github.com/gropaul/duckdb/commits/fact-intersection-join-ht`

flattens the result after the second join is computed. This probe-expansion decomposition would allow factorized filtering.

Further, it would be interesting to conduct a complexity study to evaluate whether our proposed strategy to compute cyclic joins has the same worst-case runtime guarantees as classical WCOJ algorithms. Birler et al. [19] conducted such a study for their implementation of the Expand3 operator, which is very similar to our proposed strategy. The authors showed that most cyclic queries can be decomposed in ternary cycles using binary joins and then computed with the `Expand3` operator, which has the same runtime guarantees as WCOJ joins [19].

In this work, the query optimizer rule that handles which operators should emit or consume factorized vectors was very basic. The rule opted always to use factorization if there was an operator that could emit factorized vectors, like a Hash Join Operator, and if there was an operator on top that could deal with these factorized vectors. If this is the case, the optimizer would alter the logical operators so that their physical counterparts would respectively emit or process the factors.

In regards to query optimization, the whole process of detecting (sub-) queries that could benefit from factorization, marking them as potential factorizable, and then gathering runtime metrics to make a delayed, but informed decision is still missing. It is especially interesting how to integrate the switch of plans. This requires changing the upstream operators. Here, systems that do not rely on code generation for query execution could have an advantage.

While gathering test data for the adaptive factorization and running benchmarks, we discovered that both Kùzu and Umbra made wrong decisions in regards to using WCOJs. In the example query for the evaluation of the DuckDB's factorized cyclic join performance in Listing 4.8, Umbra opted for using the WCOJ while its binary join is faster for this query. Also, Kùzu used a binary join plan while its WCOJ join plan would have been faster. Both systems required hints to use the best strategy. Therefore, we have at least anecdotal evidence that planning WCOJ and factorization is still an open research question for these systems.

To explore whether Umbra or Kùzu could benefit from adaptive decision-making, we can apply the same evaluation techniques we used to test DuckDB's factorized execution. A straightforward approach would involve benchmarking both the WCOJ and the binary join, as well as their respective optimizer, and then using the metrics obtained from these runs, similar to what we did with DuckDB, to develop a predictor that determines the optimal scenarios for using Umbra's or Kùzu's WCOJ. We could continue using the features collected during DuckDB's factorized runs as our feature set, assuming techniques like runtime sketch creation and pipelined execution can be translated to the other systems.

One big problem our proposed approach could have is overfitting. While we tried to have diverse training and test data to build our models, other users might have totally different data and workloads, on which the accuracy of our models might decline. To increase the robustness of our models, future work should focus on getting more heterogeneous data and analyzing which features might be prone to overfitting. For example, specific metrics like cardinalities might be too specific to our data. This may be due to the lesser parallelization efficiency of the factorized method compared to the binary approach, but still needs further analysis.

7 Conclusion

To answer our research question on how general-purpose systems could implement WCOJ algorithms and factorized representations, we developed an adaptive method of using collision-free hash table chains as factorized representations. This method, which we proposed, implemented, and evaluated in DuckDB, allows for the factorized computation of aggregates and, worst-case optimal cyclic join computation.

We improved the hash join operator by implementing a new method for resolving duplicates and collisions using our proposed *linear-chained* hash table. This hash table features collision-free chains by a combination of linear probing and uses micro bloom filters ("salt") at the head of the 64-bit pointers. Our implementation of this new hash table layout in DuckDB resulted in a 12% reduction in total runtime for the TPC-H benchmark and a 32% reduction for the TPC-DS benchmark at a scale factor of 100. Even at lower scale factors, we observed performance improvements, although the benefits diminish with smaller data sizes. This demonstrates that, with the salt optimization, our linear-chained hash table outperforms default hash tables. Consequently, this new hash table layout has been integrated into DuckDB, starting from version 1.1.

We then used these collision-free hash table chains as factorized representations. More specifically, this approach features what the theory of factorization calls d-representations, which are factor definitions that can be referenced by multiple tuples. We did this by emitting pointers to the hash table chains while probing instead of expanding the result directly during probing.

These d-representations were then utilized to (1) efficiently calculate aggregates and (2) perform cyclic joins in a worst-case optimal manner. For both aggregates and cyclic joins, we enhanced performance by avoiding the expansion of large intermediate results and by caching results and helper structures alongside the hash chain pointer. When a specific operation is executed on a tuple within a d-representation, other tuples sharing the same d-representation can reuse the result, eliminating the need to recompute the operation. With this approach, we achieved speedups of up to 17.58x for aggregate computations and up to 16.77x for cyclic join computations.

We then observed that these optimizations are only beneficial if there are long hash table chains that can be avoided to be expanded. To only use our optimization when beneficial, we proposed, implemented, and evaluated an adaptive way of using factorization and worst-case optimal join processing. Our approach enables adaptivity by not requiring separate operators or data structures for WCOJ and factorized representations of other systems like Kùzu or FDB. Instead, as our approach leverages a lot of existing technologies, namely hash table collision chains, factorized and non-factorized execution starts very similarly. We can, therefore, leverage the common operations the two execution strategies have and collect statistics on the data. Only when the two strategies' execution differs can we adaptively decide which plan to follow. To make this decision, we proposed machine learning classifiers. We demonstrated that these models can achieve an accuracy of up to 89%. By employing these models, we gained a speedup of 1.54x, very close to the theoretical maximum of 1.58x for our benchmark. In contrast, relying solely on factorization would result in a speedup of just 1.47x. Therefore, we showed that we can leverage nearly the full potential of adaptive factorization with adaptive decision-making and runtime metrics.

References

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. "Column-oriented database systems." In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1664–1665. URL: https://dl.acm.org/doi/10.14778/1687553.1687625 (visited on 05/22/2024) (6).

[2] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Re. "Level-Headed: A Unified Engine for Business Intelligence and Linear Algebra Querying." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE).* ISSN: 2375-026X. Apr. 2018, pp. 449–460. URL: https://ieeexplore.ieee.org/document/8509269 (visited on 08/12/2024) (37).

[3] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. *Empty-Headed: A Relational Engine for Graph Processing.* arXiv:1503.02368 [cs]. Jan. 2017. URL: http://arxiv.org/abs/1503.02368 (visited on 08/12/2024) (28, 39).

[4] Veronika Abramova and Jorge Bernardino. "NoSQL databases: MongoDB vs cassandra." In: *Proceedings of the International C* Conference on Computer Science and Software Engineering.* C3S2E '13. New York, NY, USA: Association for Computing Machinery, July 2013, pp. 14–22. URL: https://doi.org/10.1145/2494444.2494447 (visited on 06/10/2024) (6).

[5] Kirti Aggarwal and Harsh K. Verma. "Hash_RC6 — Variable length Hash algorithm using RC6." In: *2015 International Conference on Advances in Computer Engineering and Applications.* Mar. 2015, pp. 450–456. URL: https://ieeexplore.ieee.org/document/7164747 (visited on 06/11/2024) (17).

[6] Noga Alon, Yossi Matias, and Mario Szegedy. "The Space Complexity of Approximating the Frequency Moments." In: *STOC.* ACM, 1996, pp. 20–29 (99).

[7] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. *Distributed Evaluation of Subgraph Queries Using Worstcase Optimal LowMemory Dataflows.* arXiv:1802.03760 [cs]. Feb. 2018. URL: http://arxiv.org/abs/1802.03760 (visited on 08/12/2024) (37, 38, 46).

[8] Renzo Angles. "The Property Graph Database Model." In: *AMW.* 2018 (24).

[9] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. “Foundations of Modern Query Languages for Graph Databases.” In: *ACM Computing Surveys* 50.5 (Sept. 2017), 68:1–68:40. URL: `https://dl.acm.org/doi/10.1145/3104031` (visited on 06/13/2024) (24).

[10] *Apache Cassandra | Apache Cassandra Documentation*. en. URL: `https://cassandra.apache.org/_/index.html` (visited on 06/12/2024) (6).

[11] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. “Design and Implementation of the LogicBlox System.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1371–1382. URL: `https://doi.org/10.1145/2723372.2742796` (visited on 08/13/2024) (38, 46).

[12] *atomic_compare_exchange_weak, atomic_compare_exchange_strong, atomic_compare_exchange_weak_explicit, atomic_compare_exchange_strong_explicit - cppreference.com*. URL: `https://en.cppreference.com/w/c/atomic/atomic_compare_exchange` (visited on 06/26/2024) (66).

[13] Albert Atserias, Martin Grohe, and Dániel Marx. *Size Bounds and Query Plans for Relational Joins*. arXiv:1711.03860 [cs]. Nov. 2017. URL: `http://arxiv.org/abs/1711.03860` (visited on 08/11/2024) (26).

[14] RJ Atwal, Peter A Boncz, Ryan Boyd, Antony Courtney, Till Döhmen, Florian Gerlinghoff, Jeff Huang, Joseph Hwang, Raphael Hyde, Elena Felder, et al. “MotherDuck: DuckDB in the cloud and in the client.” In: *CIDR*. 2024 (33).

[15] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. “FDB: a query engine for factorised relational databases.” In: *Proceedings of the VLDB Endowment* 5.11 (July 2012), pp. 1232–1243. URL: `https://doi.org/10.14778/2350229.2350242` (visited on 04/10/2024) (30, 39, 40, 47).

[16] B. Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Y. Kiran, Sainath Mallidi, B. McGaughy, M. Personick, K. Rajan, Simone Rondelli, A. Ryazanov, Michael Schmidt, Kunal Sengupta, B. Thompson, D. Vaidya, and S. Wang. “Amazon Neptune: Graph Data Management in the Cloud.” In: 2018 (1, 7).

[17] *Benchmarks*. en. URL: `https://duckdb.org/docs/guides/performance/benchmarks.html` (visited on 06/30/2024) (70).

[18] Steve Bertolani. *Dovetail Join · RelationalAI*. en. Nov. 2021. URL: `http://localhost:4321/resources/dovetail-join/` (visited on 08/13/2024) (35).

[19] Altan Birler, Alfons Kemper, and Thomas Neumann. “Robust Join Processing with Diamond Hardened Joins.” In: 2024 (39, 43–45, 50, 108, 109).

[20] Peter Boncz, M. Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005* (Jan. 2005) (6, 32).

[21] S. Ceri, G. Gottlob, and L. Tanca. “What You Always Wanted to Know About Datalog (And Never Dared to Ask).” In: *IEEE Trans. on Knowl. and Data Eng.* 1.1 (1989), pp. 146–166. URL: `https://doi.org/10.1109/69.43410` (visited on 08/13/2024) (38).

[22] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A structured English query language.” In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control.* SIGFIDET ’74. New York, NY, USA: Association for Computing Machinery, 1974, pp. 249–264. URL: `https://dl.acm.org/doi/10.1145/800296.811515` (visited on 06/07/2024) (10).

[23] CMU Database Group and Andy Pavlo. *Hash Tables (CMU Intro to Database Systems / Fall 2021).* Sept. 2021. URL: `https://www.youtube.com/watch?v=f71kc4osCyM` (visited on 06/11/2024) (15, 17–19).

[24] CMU Database Group and Andy Pavlo. *Join Algorithms (CMU Intro to Database Systems / Fall 2022).* Oct. 2022. URL: `https://www.youtube.com/watch?v=yFk_GfaY2Hk` (visited on 06/10/2024) (20, 22).

[25] CMU Database Group, Mark Raasveldt, and Andy Pavlo. *DuckDB Internals (CMU Advanced Databases / Spring 2023).* Apr. 2023. URL: `https://www.youtube.com/watch?v=bZOvAKGkzpQ` (visited on 06/26/2024) (58).

[26] E. F. Codd. “A relational model of data for large shared data banks.” In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387. URL: `https://dl.acm.org/doi/10.1145/362384.362685` (visited on 06/07/2024) (7, 10).

[27] Edgar F Codd. “Further normalization of the data base relational model.” In: *Data base systems* 6 (1972). Publisher: Prentice-Hall Englewood Cliffs, NJ, pp. 33–64 (7).

[28] Edgar F Codd et al. *Relational completeness of data base sublanguages.* IBM Corporation, 1972 (10).

[29] *Cyan4973/xxHash: Extremely fast non-cryptographic hash algorithm.* URL: `https://github.com/Cyan4973/xxHash` (visited on 06/11/2024) (17).

[30] *DB-Engines Ranking.* en. URL: `https://db-engines.com/en/ranking` (visited on 06/07/2024) (7).

[31] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. “TigerGraph: A Native MPP Graph Database.” In: *ArXiv* (Jan. 2019). (Visited on 05/22/2024) (7).

[32] *duckdb/duckdb: DuckDB is an analytical in-process SQL database management system.* URL: `https://github.com/duckdb/duckdb` (visited on 06/14/2024) (33).

[33] *FDB Research.* URL: `https://fdbresearch.github.io/principles/second-example.html` (visited on 04/09/2024) (29–31).

[34] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. “KÙZU graph database management system.” In: *The Conference on Innovative Data Systems Research.* 2023 (7, 25, 35, 39, 41, 46–49, 92, 93).

[35] Daniel Flachs, Magnus Müller, and Guido Moerkotte. “The 3D hash join: Building on non-unique join attributes.” de. In: Chaminade: CIDR, 2022, pp. 1–9. URL: `https://madoc.bib.uni-mannheim.de/62365/` (visited on 01/08/2024) (39, 43, 45, 47, 50, 55, 57, 58, 81, 108).

[36] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. “HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm.” In: *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH,... (Mar. 2012) (102).

[37] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. “Cypher: An Evolving Query Language for Property Graphs.” In: *Proceedings of the 2018 International Conference on Management of Data.* SIGMOD ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1433–1445. URL: `https://dl.acm.org/doi/10.1145/3183713.3190657` (visited on 06/13/2024) (24).

[38] M. Freitag, Maximilian Bandle, Tobias Schmidt, A. Kemper, and Thomas Neumann. “Combining Worst-Case Optimal and Traditional Binary Join Processing.” In: 2020. (Visited on 08/14/2024) (37).

[39] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting worst-case optimal joins in relational database systems.” In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 1891–1904. URL: `https://doi.org/10.14778/3407790.3407797` (visited on 08/12/2024) (28, 36, 37, 48, 50).

[40] *google/cityhash.* original-date: 2015-08-14T21:08:16Z. June 2024. URL: `https://github.com/google/cityhash` (visited on 06/11/2024) (17).

[41] *google/farmhash.* original-date: 2015-08-14T21:01:50Z. May 2024. URL: `https://github.com/google/farmhash` (visited on 06/11/2024) (17).

[42] G. Graefe. “Volcano - An Extensible and Parallel Query Evaluation System.” In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135. URL: `https://doi.org/10.1109/69.273032` (visited on 08/14/2024) (12).

[43] Goetz Graefe. “New algorithms for join and grouping operations.” en. In: *Computer Science - Research and Development* 27.1 (Feb. 2012), pp. 3–27. URL: `https://doi.org/10.1007/s00450-011-0186-9` (visited on 06/10/2024) (15).

[44] Andrey Gubichev. “Query Processing and Optimization in Graph Databases.” PhD thesis. Technische Universität München, 2015. URL: `https://mediatum.ub.tum.de/1238730` (visited on 06/13/2024) (23).

[45] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. *Columnar Storage and List-based Processing for Graph Database Management Systems.* arXiv:2103.02284 [cs]. Oct. 2021. URL: `http://arxiv.org/abs/2103.02284` (visited on 04/09/2024) (41–43).

[46] Richard Hipp. *SQLite Home Page.* URL: `https://www.sqlite.org/` (visited on 05/21/2024) (1, 6).

[47] *Information technology — Database languages SQL - Part 16: Property Graph Queries (SQL/PGQ).* Standard. International Organization for Standardization, Mar. 2023 (24).

[48] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. “Faster set intersection with SIMD instructions by reducing branch mispredictions.” In: *Proc. VLDB Endow.* 8.3 (Nov. 2014), pp. 293–304. URL: `https://doi.org/10.14778/2735508.2735518` (visited on 08/13/2024) (39).

[49] Prashant Kannan, Chuck Murray, Melliyal Annamalai, Korbinian Schmid, Albert Godfrind, Oskar van Rest, Jorge Barba, Ana Estrada, Steve Serra, Ryota Yamanaka, Bill Beauregard, Hector Briseno, Hassan Chafi, Eugene Chong, Souripriya Das, Juan Garcia, Florian Gratzer, Zazhil Herena, Sungpack Hong, Roberto Infante, Hugo Labra, Gabriela Montiel-Moreno, Eduardo Pacheco, Joao Paiva, Matthew Perry, Diego Ramirez, Siva Ravada, Carlos Reyes, Jane Tao, Edgar Vazquez, Zhe (Alan) Wu, and Lavanya Jayapalan. *What Are Property Graphs?* en-US. concept. Publisher: June2022. URL: `https://docs.oracle.com/en/database/oracle/property-graph/22.2/spgdg/what-are-property-graphs.html` (visited on 06/13/2024) (24).

[50] Alfons Kemper and Thomas Neumann. “HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots.” In: *2011 IEEE 27th International Conference on Data Engineering.* ISSN: 2375-026X. Apr. 2011,

pp. 195–206. URL: https://ieeexplore.ieee.org/document/5767867 (visited on 08/12/2024) (37).

[51] Russ Kennedy. *The New Era Of Big Data.* en. Section: Innovation. May 2023. URL: https://www.forbes.com/councils/forbestechcouncil/2023/05/24/the-new-era-of-big-data/ (visited on 08/17/2024) (1).

[52] Laurens Kuiper, Peter Boncz, and Hannes Mühleisen. "Robust External Hash Aggregation in the Solid State Age." English. In: IEEE Computer Society, May 2024, pp. 3753–3766. URL: https://www.computer.org/csdl/proceedings-article/icde/2024/171500d753/1YOtBb8JI7C (visited on 08/14/2024) (33, 67).

[53] Jérôme Kunegis. "KONECT: the Koblenz network collection." In: *Proceedings of the 22nd International Conference on World Wide Web.* WWW '13 Companion. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1343–1350. URL: https://doi.org/10.1145/2487788.2488173 (visited on 05/23/2024) (43).

[54] Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines." In: *Proceedings of the 14th International Workshop on Data Management on New Hardware.* DAMON '18. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 1–8. URL: https://doi.org/10.1145/3211922.3211928 (visited on 08/15/2024) (32).

[55] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* SIGMOD '14. New York, NY, USA: Association for Computing Machinery, June 2014, pp. 743–754. URL: https://dl.acm.org/doi/10.1145/2588555.2610507 (visited on 02/05/2024) (6, 32, 36).

[56] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases." In: *2013 IEEE 29th International Conference on Data Engineering (ICDE).* ISSN: 1063-6382. Apr. 2013, pp. 38–49. URL: https://ieeexplore.ieee.org/document/6544812 (visited on 06/14/2024) (33).

[57] Daniel Lemire. *A fast alternative to the modulo reduction – Daniel Lemire's blog.* en-US. June 2016. URL: https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/ (visited on 06/28/2024) (63).

[58] Daniel Lemire. "Fast Random Integer Generation in an Interval." In: *ACM Transactions on Modeling and Computer Simulation* 29.1 (Jan. 2019). arXiv:1805.10941 [cs], pp. 1–12. URL: http://arxiv.org/abs/1805.10941 (visited on 06/28/2024) (63).

[59] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. “Graphs over time: densification laws, shrinking diameters and possible explanations.” In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. KDD ’05. New York, NY, USA: Association for Computing Machinery, Aug. 2005, pp. 177–187. URL: `https://doi.org/10.1145/1081870.1081893` (visited on 05/23/2024) (42).

[60] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014 (97).

[61] Dapeng Liu, Zengdi Cui, Shaochun Xu, and Huafu Liu. “An empirical study on the performance of hash table.” In: *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*. June 2014, pp. 477–484. URL: `https://ieeexplore.ieee.org/abstract/document/6912180` (visited on 06/11/2024) (19).

[62] Andrew Mayers. *CS 312: Hash tables and amortized analysis*. Cornell University, Department of Computer Science., Apr. 2021. URL: `https://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec20.html` (visited on 06/26/2024) (62).

[63] Kurt Mehlhorn and Peter Sanders. *Algorithms and data structures: the basic toolbox*. eng. Berlin Heidelberg: Springer, 2008 (17, 18).

[64] Puya Memarzia, Suprio Ray, and Virendra C Bhavsar. “On Improving Data Skew Resilience In Main-memory Hash Joins.” In: *Proceedings of the 22nd International Database Engineering & Applications Symposium*. IDEAS ’18. New York, NY, USA: Association for Computing Machinery, June 2018, pp. 226–235. URL: `https://doi.org/10.1145/3216122.3216156` (visited on 06/17/2024) (55).

[65] Amine Mhedhbi. “GraphflowDB: Scalable Query Processing on Graph-Structured Relations.” en. Accepted: 2023-10-02T14:39:48Z. Doctoral Thesis. University of Waterloo, Oct. 2023. URL: `https://uwspace.uwaterloo.ca/handle/10012/19981` (visited on 01/07/2024) (26, 28, 30, 35–39, 42, 48, 49).

[66] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. “LSQB: a large-scale subgraph query benchmark.” In: *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA ’21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 1–11. URL: `https://dl.acm.org/doi/10.1145/3461837.3464516` (visited on 06/13/2024) (25).

[67] Priti Mishra and Margaret H. Eich. “Join processing in relational databases.” In: *ACM Computing Surveys* 24.1 (1992), pp. 63–113. URL: https://dl.acm.org/doi/10.1145/128762.128764 (visited on 06/10/2024) (10, 14).

[68] Guido Moerkotte and Thomas Neumann. “Accelerating queries with group-by and join by groupjoin.” In: *Proceedings of the VLDB Endowment* 4.11 (Aug. 2011), pp. 843–851. URL: https://dl.acm.org/doi/10.14778/3402707.3402723 (visited on 04/08/2024) (85).

[69] Alex Monahan. *Even Friendlier SQL with DuckDB*. en. Aug. 2023. URL: https://duckdb.org/2023/08/23/even-friendlier-sql.html (visited on 06/14/2024) (33).

[70] *MongoDB: The Developer Data Platform*. en-us. URL: https://www.mongodb.com (visited on 06/12/2024) (6).

[71] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. “Naiad: a timely dataflow system.” In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP ’13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 439–455. URL: https://dl.acm.org/doi/10.1145/2517349.2522738 (visited on 08/12/2024) (37).

[72] Raghunath Othayoth Nambiar and Meikel Poess. “The making of TPC-DS.” In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB ’06. Seoul, Korea: VLDB Endowment, Sept. 2006, pp. 1049–1058. (Visited on 06/30/2024) (70).

[73] Ovais Naseem. *Relational vs. Non-Relational Databases - DataScienceCentral.com*. en-US. Feb. 2024. URL: https://www.datasciencecentral.com/decoding-different-types-of-databases-a-comparison/ (visited on 06/10/2024) (6).

[74] *Neo4j Graph Database & Analytics – The Leader in Graph Databases*. en. URL: https://neo4j.com/ (visited on 05/22/2024) (1, 7).

[75] *NetworkX — NetworkX documentation*. URL: https://networkx.org/ (visited on 08/09/2024) (97).

[76] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance.” In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf (visited on 05/22/2024) (36, 46–48, 92, 93).

[77] Hung Q Ngo, Christopher Ré, and Atri Rudra. “Skew strikes back: new developments in the theory of join algorithms.” In: *SIGMOD Rec.* 42.4 (Feb. 2014), pp. 5–16. URL: https://dl.acm.org/doi/10.1145/2590989.2590991 (visited on 08/12/2024) (27, 28, 35, 43, 44).

[78] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. *Worst-case Optimal Join Algorithms.* arXiv:1203.1952 [cs, math]. Mar. 2012. URL: http://arxiv.org/abs/1203.1952 (visited on 05/14/2024) (14, 27).

[79] Dan Olteanu. “Factorized Databases: A Knowledge Compilation Perspective.” In: 2016. (Visited on 04/10/2024) (30, 41).

[80] Dan Olteanu and Jakub Zavodny. *Factorised Representations of Query Results.* arXiv:1104.0867 [cs]. Apr. 2011. URL: http://arxiv.org/abs/1104.0867 (visited on 04/09/2024) (80).

[81] Dan Olteanu and Jakub Závodný. “Size Bounds for Factorised Representations of Query Results.” In: *ACM Transactions on Database Systems* 40.1 (2015), 2:1–2:44. URL: https://doi.org/10.1145/2656335 (visited on 05/14/2024) (31, 81).

[82] Karl Pearson and Francis Galton. “Note on regression and inheritance in the case of two parents.” In: *Proceedings of the Royal Society of London* 58.347-352 (1895). Publisher: Royal Society, pp. 240–242. URL: https://royalsocietypublishing.org/doi/10.1098/rspl.1895.0041 (visited on 08/09/2024) (101).

[83] *PyPI Download Stats.* URL: https://pypistats.org/packages/duckdb (visited on 05/22/2024) (1, 6).

[84] Mark Raasveldt and Hannes Mühleisen. “DuckDB: an Embeddable Analytical Database.” In: *Proceedings of the 2019 International Conference on Management of Data.* SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 1981–1984. URL: https://doi.org/10.1145/3299869.3320212 (visited on 05/21/2024) (1, 6, 33, 92, 93).

[85] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems.* McGraw-Hill, Inc., 2002 (1, 17).

[86] Pingan Ren. “Parallelized Path-finding in DuckPGQ.” MA thesis. VU Amsterdam, July 2024. URL: https://homepages.cwi.nl/~boncz/msc/2023-ThomasGlas.pdf (visited on 08/17/2024) (34).

[87] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases: new opportunities for connected data.* ” O’Reilly Media, Inc.”, 2015 (23).

[88] Kenneth A. Ross. “Efficient Hash Probes on Modern Processors.” In: *2007 IEEE 23rd International Conference on Data Engineering.* ISSN: 2375-026X. Apr. 2007, pp. 1297–1301. URL: `https://ieeexplore.ieee.org/document/4221787` (visited on 06/28/2024) (18, 63).

[89] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. “The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey.” In: *The VLDB Journal* 29.2-3 (May 2020). arXiv:1709.03188 [cs], pp. 595–618. URL: `http://arxiv.org/abs/1709.03188` (visited on 05/22/2024) (1, 26, 48).

[90] Kim Shanley. *History of TPC.* Feb. 1998. URL: `https://www.tpc.org/information/about/history5.asp` (visited on 06/30/2024) (70).

[91] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts.* Seventh edition. New York, NY: McGraw-Hill, 2020 (5, 7, 9, 10, 12, 14, 15, 17, 20).

[92] *std::unordered_map - cppreference.com.* URL: `https://en.cppreference.com/w/cpp/container/unordered_map` (visited on 06/12/2024) (18).

[93] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. “Birthday Paradox for Multi-collisions.” en. In: *Information Security and Cryptology – ICISC 2006.* Ed. by Min Surp Rhee and Byoungcheon Lee. Berlin, Heidelberg: Springer, 2006, pp. 29–40 (18).

[94] Gábor Szárnyas et al. “The LDBC Social Network Benchmark: Business Intelligence Workload.” In: *Proc. VLDB Endow.* (2022), pp. 877–890 (97).

[95] Jordan Tigani. *Big Data is Dead.* en. Mar. 2023. URL: `https://motherduck.com/blog/big-data-is-dead/` (visited on 08/17/2024) (1).

[96] *TPC-DS Homepage.* URL: `https://www.tpc.org/tpcds/` (visited on 06/30/2024) (70).

[97] *TPC-H Homepage.* URL: `https://www.tpc.org/tpch/` (visited on 06/30/2024) (70).

[98] Richard J. Trudeau and Richard J. Trudeau. *Introduction to graph theory.* Dover books on advanced mathematics. New York: Dover Pub, 1993 (23).

[99] Jeffrey D. Ullman and Jeffrey D. Ullman. *Classical database systems.* eng. 8. printing. Principles of database and knowledge-base systems / Jeffrey D. Ullman Vol. 1. Rockville, Md: Computer Science Press, 1995 (10).

[100] GitHub User. *Release Calendar.* en. URL: `https://duckdb.org/docs/dev/release_calendar.html` (visited on 08/11/2024) (107).

[101] Todd L. Veldhuizen. *Incremental Maintenance for Leapfrog Triejoin.* arXiv:1303.5313 [cs]. Mar. 2013. URL: `http://arxiv.org/abs/1303.5313` (visited on 08/13/2024) (38, 39).

[102] Todd L. Veldhuizen. *Leapfrog Triejoin: a worst-case optimal join algorithm.* arXiv:1210.0481 [cs]. Dec. 2013. URL: `http://arxiv.org/abs/1210.0481` (visited on 08/13/2024) (38).

[103] Yisu Remy Wang, Max Willsey, and Dan Suciu. "Free Join: Unifying Worst-Case Optimal and Traditional Joins." In: *Proceedings of the ACM on Management of Data* 1.2 (June 2023), 150:1–150:23. URL: `https://dl.acm.org/doi/10.1145/3589295` (visited on 01/09/2024) (29).

[104] Adrienne Watt and Nelson Eng. *Database Design, 2nd edition.* eng. Accepted: 2018-02-26T20:54:16Z. BCcampus, 2014. URL: `https://openlibrary-repo.ecampusontario.ca/jspui/handle/123456789/247` (visited on 06/07/2024) (7).

[105] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and P. Boncz. "DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS." In: 2023. (Visited on 05/14/2024) (24, 34).

[106] Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. "DuckPGQ: Bringing SQL/PGQ to DuckDB." In: *Proceedings of the VLDB Endowment* 16.12 (Aug. 2023), pp. 4034–4037. URL: `https://dl.acm.org/doi/10.14778/3611540.3611614` (visited on 06/14/2024) (34).

[107] Weipeng P. Yan and Per-Åke Larson. "Eager Aggregation and Lazy Aggregation." In: *Proceedings of the 21th International Conference on Very Large Data Bases.* VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Sept. 1995, pp. 345–357. (Visited on 08/15/2024) (44).

[108] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. "MonetDB/X100-A DBMS In The CPU Cache." In: *IEEE Data Eng. Bull.* 28.2 (2005), pp. 17–22 (32).