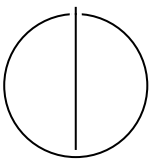
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Supporting Joins in OpenIVM

Steffano Papandroudis



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

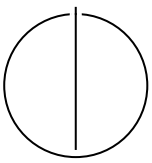
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Supporting Joins in OpenIVM

Implementierung von Joins in OpenIVM

Author:	Steffano Papandroudis
Examiner:	Prof. Dr. Thomas Neumann
Supervisor:	Prof. Dr. Peter Boncz, Ilaria Battiston M.Sc.
Submission Date:	15.05.2025



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.05.2025 Steffano Papandroudis

Acknowledgments

This thesis was written at the Centrum voor Wiskunde & Informatica (CWI) in Amsterdam, the Netherlands.

Abstract

Incremental View Maintenance (IVM) is a method of updating materialised views inside databases by processing only modified data, thus avoiding complete recomputation. OpenIVM is a DuckDB extension that acts as a SQL-to-SQL IVM implementation, designed to bring IVM capabilities to any system using SQL.

This Master's thesis extends OpenIVM by implementing incremental maintenance for inner joins, a core operation in database systems. In addition, a system to convert logical query plans into SQL was devised. Using both, the implementation of joins in OpenIVM is benchmarked and compared with a reference join implementation without IVM, demonstrating that OpenIVM outperforms the reference join implementation when less than 5% of the underlying data is updated.

Finally, the thesis discusses approaches for supporting additional join types in OpenIVM, laying the groundwork for future enhancements in adaptive and efficient view maintenance.

Contents

Part I.

Background

1. Introduction

In the field of relational databases, views provide an overview or summary of data stored inside a Relational Database Management System (RDBMS). Views are named, predefined queries that can themselves be queried by using their name as if it were an actual table. A Materialised View (MV) is a precomputed view that the system actually stores as a a table. Hence, when queried, their result is immediately available, and can thus greatly accelerate query performance.

When data is updated inside a database, an MV that depends on this data will also have to be updated. Incremental View Maintenance (IVM) ensures that a MV can be updated *incrementally* – using only the modified data as a basis – rather than to recompute the view completely from scratch.

Most current implementations of IVM are either theoretical, or specialised for a specific system. In contrast, OpenIVM aims to bring support for IVM to any system supporting Structured Query Language (SQL), by representing all operations necessary to perform IVM as SQL queries [1]. The scope of this thesis is to extend the current functionality of OpenIVM by introducing support for joins, which is one of most important operations inside any RDBMS. Joins combine data from different tables, which is a fundamental aspect of relational databases. To this end, this thesis is oriented around the following research questions:

1. How can join functionality be added to OpenIVM?
2. Under what workloads is IVM on queries with an inner join beneficial?
3. Is it viable to convert arbitrary logical query plans generated by DuckDB into SQL, and if so, how?

Based on the above questions, this thesis discusses several contributions to OpenIVM. First, a 'rewrite rule' for OpenIVM is discussed. This rewrite rule – for queries that create and maintain a materialised view – takes care of converting logical plans containing joins into a plan that that allows for incremental updates. The logical plan is first optimised by DuckDB, then modified by the OpenIVM extension. Furthermore, a *Logical Plan to SQL* converter is created from scratch inside a *Compiler* extension. This compiler converts the modified logical plan into a SQL query compatible with DuckDB

and other SQL RDBMSs using the Postgres SQL dialect. This compiler also includes a first draft for an Abstract Syntax Tree (AST), which makes it possible to easily support other SQL dialects possible in the future. Finally, the implementation is examined using benchmarks. Multiple benchmarks compare the IVM performance of views with joins to their non-IVM counterpart, to devise methods for adaptive maintenance.

The remainder of this thesis is structured as follows.

Chapters 2, 3, and 4 comprise the background of this thesis. In Chapter 2, the basics of database systems (insofar relevant to this thesis) are described. In Chapter 3, IVM, OpenIVM, and related concepts are explained and elaborated upon. Chapter 4 is dedicated to DuckDB – a database management system that has been extensively used throughout this thesis – and also covers some of its implementation details.

The three chapters thereafter are dedicated to the contributions made in this thesis. Chapter 5 discusses the implementation of joins inside DuckDB's OpenIVM extension. Chapter 6 consequently elaborates upon contributions to the `Compiler` extension of DuckDB, which converts logical query plans to SQL. In Chapter 7, benchmarks of these implementations are depicted, and their performance is compared to non-IVM queries.

Chapter 8 discusses any findings in this thesis, such as limitations of IVM join support. Finally, Section 9.2 contains some concluding remarks, including potential future work.

2. Relational Database Management Systems (RDBMS)

The purpose of this chapter is to provide the necessary background knowledge on RDBMS to understand concepts and further considerations in this thesis. This chapter therefore aims to take a holistic approach to the topic at hand, covering the concepts and notions relevant to databases and IVM.

Much of the content in this chapter is based on the seminal work of Codd [2] in the field of databases, as well as literature by Kleppmann [3] and formal standards from ISO/IEC [4].

The chapter commences with some high-level RDBMS concepts. First, Section 2.1 briefly describes the relevant aspects of database systems. Section 2.2 discusses the workloads that database systems typically have to endure, and Section 2.3 their typical query interface.

After having covered RDBMS concepts at a high level, Sections 2.4 and 2.5 both focus on notions that will be frequently referenced throughout this thesis. Section 2.4 discusses (materialised) views, whereas Section 2.5 explores the relational join operator and its several variants.

On the lower end of RDBMS concepts, Section 2.6 is dedicated to query execution inside an RDBMS.

With this knowledge at hand, it becomes possible to explore IVM in Chapter 3, and subsequently explain the specifics of DuckDB in Chapter 4.

2.1. A primer on database systems

The purpose of a database system is to efficiently store, process, and obtain data on electronic devices. Most database systems are relational, meaning that there are relationships between data points that can be explicitly defined through various different mechanisms. Data is stored in tables, which are subdivided into columns. Tables are populated by records, and each record is an individual entity. Each column has a specific data type, which may optionally be `NULL`able (meaning that a record may have no value defined for that column).

All columns of a record are inherently related to each other, simply by virtue of being part of the same record. In addition, specific columns may refer to entities stored or defined in other tables – for example by using an identifier (ID) – which in turn forms a relationship between entities across tables. Depending on the database system, this relationship may be solidified through so-called *keys*. This will be elaborated upon when joins are discussed in Section 2.5.

The concept of explicitly defined relationships between entities (or data points) is in stark contrast to conventional methods of storing data on electronic devices. Most often, operating systems store data on a per-file basis, without any sophisticated hierarchy in place. No relationships – implicit or explicit – are defined between such files on a disk, which means that data stored across several files can most often not be easily combined together for analytical purposes.

Any read or write operation in an RDBMS is done through a query (except for some configuration commands), and almost all database systems use SQL[1] for queries. SQL is a declarative language, meaning that users writing a query only need to describe *what* they want, and not *how* to achieve that. Therefore, two different queries with an identical semantic meaning should (in theory) have the same execution performance, no matter how the queries are written.[2] It must be noted that – although SQL is a declarative language – a user's decisions may nevertheless have influence on the performance of a database system. Namely, the choices made when defining a table (such as the types used for each column) as well as the inclusion or exclusion of auxiliary structures (such as indices and constraints) can optimise certain operations.

Database systems are especially useful when:

- Data needs to be analysed.
- Data needs to be in a standardised format.
- The integrity and consistency of data is important.

According to Codd [2], a user of an RDBMS should not have to know the internal representation of the data in a machine – both in format and structure. It is generally the complete responsibility of the RDBMS how and where the data is stored.[3] This is in contrast to the file system on most machines, where users are expected to organise the format, structure and hierarchy of files themselves.

[1] The acronym 'SQL' is often pronounced as 'sequel'.

[2] There are cases in which some database systems cannot successfully optimise queries written in a specific manner, for example when correlated subqueries are present. In those cases, two semantically identical queries may have a significantly different execution performance.

[3] Distributed database systems generally do require significant custom configuration on how and where data is stored, but this is beyond the scope of this thesis.

Regardless of any optimisations that an RDBMS may implement, including on the storage level, the interface for users who query the database should remain unaffected. Keeping in mind that SQL is a declarative language, it becomes possible to introduce many optimisations to an RDBMS without end users needing to be aware of them.

This concept and its advantages may become more apparent in the following subsections. Subsection 2.2 explains the different workloads that databases tend to be used for, as well as their technical constraints. Then, subsections 2.4 and 2.5 elaborate upon two particular functionalities of RDBMS: views and joins. These functionalities are essential to understand the remainder of this thesis.

2.2. Database workloads

Different database systems serve different workloads and scopes. Some database systems are intended for usage within one device only, whereas others are meant to be hosted on a server, with clients connecting to it through an internet connection. Since different database workloads can have different implications for the design, usage, and structuring of an RDBMS, the typical workloads are elaborated upon.

Broadly speaking, database workloads can be categorised into two groups: transactional workloads – often named Online Transaction Processing (OLTP) – and analytical workloads – commonly named Online Analytical Processing (OLAP). Subsection 2.2.1 covers OLTP workloads, whereas Subsection 2.2.2 covers OLAP. The design considerations that come with database workloads are then discussed in Subsection 2.2.3.

2.2.1. Online Transaction Processing (OLTP)

OLTP refers to workloads in which frequent transactions (write operations) occur, and in which lookups typically only access few records [3]. In production, an OLTP database is frequently used as the backbone of one or more applications, where insertions and updates are based on the input of some user. Commonly used OLTP RDBMSs include SQLite, MySQL, and PostgreSQL.[4]

A traditional example of an OLTP-like system would be a bank that handles (literal) transactions between different accounts. To be able to process OLTP workloads effectively, the main operation of interest for the database is to handle multiple write operations (sometimes concurrent or otherwise high in volume), while reads across the entire database are unlikely to occur often.

[4] `https://sqlite.org/`, `https://www.mysql.com/`, `https://www.postgresql.org/`.

2.2.2. Online Analytical Processing (OLAP)

OLAP conversely refers to workloads in which the analysis of data is the predominant purpose of a database. According to Kleppmann [3], databases specifically for OLAP emerged in the late 1980s. Previously, OLAP and OLTP workloads would most often share a database system. Commonly used OLAP RDBMSs and services include DuckDB, Snowflake, and Databricks.[5]

Characteristic for OLAP workloads is the prevalence of aggregations inside queries, covering a large number of records. This includes both numerical aggregate operations such as `sum` and `avg` (average), but also the grouping of data by periods of time. Typical OLAP use cases include data analytics and business intelligence applications.

Handling a high volume of updates is typically of secondary importance to OLAP systems, and there are usually different priorities when it comes to concurrency and consistency. Since queries generally are more complex and take longer to run, there is less emphasis on instant updates than for OLTP workloads. Rather than always having an up-to-date database, it may sometimes be preferred for OLAP workloads to receive any updates in batches.

2.2.3. Design considerations

The intended workload of a database can have major consequences for the design and optimisation strategy of the RDBMS. In this subsection, some of the design decisions that are relevant depending on workload are discussed.

In OLTP workloads, read-only queries tend to be highly selective. A high *selectivity* means that the amount of records needed to perform a certain query is only a small fraction of the total amount of records in an entire table. When queries are deemed to be selective, a sequential scan of the data in a record may not be the most efficient means of accessing the data needed for a query. Instead, an *index* – an advanced data structure for enhancing lookup performance – may be used to retrieve the necessary records with a significantly better runtime than a sequential scan. In mathematical terms: if a table has n records, a sequential scan has a complexity of $O(n)$ (meaning this is the runtime needed to fetch all records of interest), whereas an index lookup typically only has a complexity of $O(\log(n))$.[6] Whether it is worth using an index for a specific query depends on the selectivity and the additional overhead for using an index lookup. The overhead for record retrieval using index lookups is generally significant. Consequently, for tables with a non-trivial amount of records, the selectivity threshold for index lookups is usually around $n < 0.01$ or $n < 0.05$ – meaning that

[5] `https://duckdb.org/`, `https://www.snowflake.com/`, `https://www.databricks.com/`

[6] Note: any usages of 'log' in this thesis, except if otherwise stated, are base 2 (so equivalent to $\log_2$).

index lookups are more efficient only if a query requires at most 1% to 5% of records in a table.

Using an index comes at a cost: it requires maintenance after any modifications to the data. To determine whether it is worth adding an index to a specific data, the ratio between insertion frequency and frequency of selective queries has to be considered. In a workload with many small write operations, frequent full table scans, and only rarely a selective query, setting up and maintaining an index data structure may not be worth it. For OLAP workloads, indices are usually a less attractive option for most tables.

The different workloads are also to be taken into consideration when storing the data itself – beyond indices – on disk.

For OLTP workloads, it is beneficial to optimise for reading and/or writing all information pertaining to a specific record. Since OLAP workloads are most commonly querying aggregate data – rather than having interest in specific records – it would instead be beneficial to optimise for processing all values pertaining to a specific database column [3].

One possible means of optimising for OLAP workloads is by using columnar storage. When columnar storage is used, it becomes significantly more efficient (relative to record-based storage) to read all values of a database column. This comes at the cost of a lower efficiency when all columns of specific records need to be accessed – a scenario which occurs more often in OLTP workloads.

Beyond storage and data structures, there are also optimisations to be made on the execution of the queries themselves. Whereas an OLTP workload often has to be optimised for concurrency, OLAP can benefit from reducing the execution time of complex queries. Such complex queries often involve vast amounts of data, and therefore there are many potential bottlenecks that can affect performance.

The most demanding tasks in complex queries usually involve sorting and aggregation, meaning that enhancements for these operations can have a great impact on the overall performance. A possible means of optimisation is to increase the throughput of computations to the Central Processing Unit (CPU) of a system, for example using vectorisation techniques [5]. In short, vectorisation is a parallel processing technique which leverages the Single Instruction, Multiple Data (SIMD) capabilities of contemporary CPUs. Since the CPU can process multiple values concurrently per CPU instruction (and therefore cycle), any algorithms that support vectorised execution require significantly fewer CPU cycles than their non-vectorised counterparts. Since data (physically) stored in columns is straightforward to vectorise, vectorised execution and columnar data storage are mutually beneficial.

It must be noted that the aforementioned optimisations are not the only means of optimising for OLAP workloads. OLAP queries may also benefit from code generation

enhancements – especially when combined with memory optimisations – which is a technology heavily used by (for example) the Umbra RDBMS [6].

Finally, a brief tangent must be made to distributed database systems. So far, any discussion related to RDBMS optimisation implicitly assumed that the RDBMS is local to (or *embedded* into) one physical system. When a database becomes large enough that scaling out (using multiple systems) is inevitable, more factors have to be considered for optimisation. For a distributed RDBMS, choices have to be made on matters such as data distribution, latency reduction, and consensus algorithms. Although distributed systems are beyond the scope of this thesis, this notion is important to mention for some later concepts in Section 3.2.

To conclude, optimising RDBMSs is a multifaceted process, in which every component of a system (as well as its workload) can matter.

2.3. Querying a database using Structured Query Language (SQL)

As mentioned in Section 2.1, SQL is a declarative language used to query the vast majority of present-day database systems. The language is formalised by the International Organization for Standardization (ISO), under ISO/IEC 9075, with the most recent standard being published in 2023 [4].

Although the language has a formal standard, different database systems often make some minor alterations to the language when implementing it for their RDBMS. These derivations from the SQL-standard are commonly named *dialects*, and usually concern secondary functionality of the language. The main differences across dialects usually become apparent when declaring a table or when casting types, as the support and storage of data types tends to vary across systems.

2.3.1. A basic SQL query

For illustrational purposes, Listing 2.1 depicts a typical SQL-query. In natural language, this query 'requests' the following from the database: "Give all records from `table_1` for which the value of the `age` column is greater than 42, but exclude those whose `name` is 'Foo'. The output should consist of the `name` and `age` columns of the table."

```
1 SELECT name, age
2 FROM table_1
3 WHERE age > 42 AND name != 'Foo';
```

Listing 2.1: A simple SQL query.

The user has no direct influence on how this query is executed. Internally, the database converts the query into a *logical plan*, in which each keyword roughly corresponds to an operator. Further details on this are discussed in Section 2.6.

2.3.2. Aggregations and joins

Beyond this simple example, SQL has further commonly-used keywords and features to query a database. A `GROUP BY` clause (formally called an aggregation) makes it possible to group records together based on a certain property, and to describe this group with a statistic. When considering the previous example (Listing 2.1), it would be possible to create a query yielding the amount of people by age. Such a query would look something like Listing 2.2.

```
1 SELECT age, COUNT(*)
2 FROM table_1
3 GROUP BY age
4 ORDER BY age DESC;
```

Listing 2.2: An example aggregation query in SQL.

This query also contains another keyword: `ORDER BY`. This keyword makes it possible to sort the output of a query. In this case, the output is sorted by the age in descending order.

Perhaps the most distinctive feature of relational databases is the concept of *joins*. Joins are used to combine data from two different tables into one result set, based on a condition that pairs records from both tables. In SQL, one can implicitly use a join in their query by specifying two tables (in the `FROM`-clause) and specifying a condition using both tables in the `WHERE`-clause. There are multiple join variants which can be called either implicitly or explicitly. This is discussed in further detail in Section 2.5.

When no conditions are specified, and no explicit join type is specified, all records from one table are joined with all records from the other table, yielding a *cross product*. Cross products are highly inefficient for non-trivial amounts of records, and are therefore avoided in query execution whenever possible.

2.3.3. Data manipulation

Data manipulation through SQL is generally done using the `INSERT`, `UPDATE`, and `DELETE` statements. Their usage is fairly straightforward: insertions are done using either constant values or any arbitrary `SELECT` statement. In either case, it is important that the column count in the statement matches the columns of the database table to update[7], and that each individual column has a matching type. Updates and deletions function similarly. Insertions and updates have some additional syntax for conflict resolution, which becomes relevant when constraints on a table allow a specific key to only occur once.

2.3.4. Common Table Expressions (CTEs)

Finally, queries can contain nested queries and – as such – can become arbitrarily complex. Rather than writing one long query or a query with multiple levels of nesting, it is possible to write a query as a series of smaller *subqueries*, so that the query can be written and interpreted by a human more easily. SQL offers an ergonomic syntax to write down such subqueries in a readable manner, namely by putting each subquery inside a Common Table Expression (CTE). A CTE is a type of subquery – to be declared before the main body of a query – which yields an intermediate named result set. An example is depicted in Listing 2.3.

Since each CTE is given a name, the remainder of the query can be written as if the CTE were a table. A CTE is not a physical table, and as such, is not usable outside the query it is declared in. CTEs are therefore purely a syntactical aid for writing a query.

```
1 WITH senior_employees AS (
2   SELECT * FROM table_1
3   WHERE age > 42
4 ),
5 no_foo AS (
6  SELECT * FROM senior_employees
7  WHERE name != 'Foo'
8 )
9 SELECT name, age FROM no_foo;
```

Listing 2.3: An example query using two CTEs. The second CTE (`no_foo`) makes use of the first CTE (`senior_employees`).

[7] This is not necessarily the amount of columns in the table itself. For example, some database systems permit the omission of columns with an auto-incrementing index (alternatively named *identity columns*).

2.4. Views in database systems

In an RDBMS, a *view* is essentially a query whose response is output as if it is a table. The view can be called using its name – similar to how tables have a name – effectively making a view a virtual table. However, since a view persists after its initial query, it is expressly not synonymous to a CTE.

In principle, it is not possible to directly modify the data (insert, update, delete) in a view.[8] However, if one updates the data in the tables based on which a view is created, a query on the view should (eventually) reflect these updates as well. Aside from base table records, database views can also make use of aggregate functions – for example `avg()` (average) or `sum()` – making them useful for analytics.

Database views can be divided into roughly two behaviours: a (normal) view and a MV. Whereas normal views merely query the data from the base tables every time a user queries the view, an MV stores the query result as a table. As a consequence, MVs have to maintain the content of the view by keeping track of changes made to the base tables upon which the view is based. If changes are made to the base table(s) that a view uses, then (eventually) these changes should also be materialised for the view. As a special case, it must be noted that MVs may also be stale, which means that they are not updated upon their creation.

```
1  -- View
2  CREATE VIEW age_histogram AS
3    SELECT age, COUNT(*)
4    FROM table_1
5    GROUP BY age;
6
7  -- Materialised view
8  CREATE MATERIALIZED VIEW age_histogram AS
9    SELECT age, COUNT(*)
10   FROM table_1
11   GROUP BY age;
```

Listing 2.4: An example view (and materialised view) in SQL, which yields the amount of records per age in `table_1`.

[8] Strictly speaking, the SQL standard permits write operations (such as `INSERT`) on a view, albeit under narrow conditions [4]. Namely, direct updates are allowed only if there exists an injective relationship between records in the view and records in one specific base table, such that each record in a view unambiguously maps to a record in the base table. Consequently, this excludes any queries using aggregates or set operations, as well as queries using more than one base table. Some database systems, notably embedded systems such as SQLite [7], indiscriminately forbid direct updates to views.

In SQL, a view can be defined as depicted in Listing 2.4 – which also shows an equivalent query for an MV.

To update MVs, several methods exist. One method would be to instantly update the MV whenever a change is detected in the base table(s). This guarantees that the view is always up to date, although this may add a non-trivial maintenance overhead, and therefore may make the database less performant. An alternative method would be to update the view only when it is actually queried. This means that the database only checks whether a view is up-to-date whenever it is called, rather than proactively keeping track of the changes made to the base table(s). This reduces the load on the database, and ensures that the view is never updated more often than it is called. However, this does mean that the individual queries made on the view may take longer to yield an output. Finally, the materialisation of the view might be detached from any query execution (whether on the base tables or the view itself), and instead take place on a periodic basis, for example every 5 minutes. In this situation, the output of the view is not guaranteed to be completely up-to-date, but can nevertheless assure that the data is relatively recent. More specifically, it can be assured that the view represents either the state of the base tables at present, or the state at *some* moment in the recent past [8]. Refreshes may be triggered by job schedulers such as `cron`.

Of course, one may also devise hybrid tactics, such as combining a time-based recency with a maximum amount of row updates before an update on the view is forced.

In the end, the preferred MV refresh strategy depends on the RDBMS. In particular, aspects such as accuracy, speed, and processing power have to be considered.

2.5. Relational joins

In this section, joins are explored in greater detail. As mentioned in Section 2.3, a join creates *pairs* of records from different tables based on some condition. A condition is defined using a predicate. Example join predicates include equality ($=$) and inequality ($!=$ or $<>$), comparisons such as $>$ or $\leq$, but also some SQL-specific predicates to handle `NULL`s such as '`IS DISTINCT FROM`'.

2.5.1. Overview of join types

For convenience, a short summary of various join types is provided below, where L and R are both base tables (or relations in the context of Relational Algebra (RA)) and $|L|$ and $|R|$ the amount of records in their respective tables:

Inner join ($L \bowtie R$) A join between L and R for which any provided conditions must hold. The result set is equal to the amount of matching pairs between L and R.

Equi-join ($L \bowtie_{L.x=R.y} R$) A special variant of the inner join, in which the join condition specifies that the value of a specific column in L must be equal to a specific column in R. An inner join may also be called a natural join when two sides can be implicitly joined by one or more identically named columns.

Left outer join ($L \mathbin{⟕} R$) A join between L and R, in which each tuple in L appears at least once in the result set. If a record in L can find one or more matches in R, then this record will appear once in the result set for each match made with a record for R. If no match can be made with any record in R, the record will still be part of the result set, but with the columns corresponding to R kept vacant (that is, all such columns have the value `NULL`).

Right outer join ($L \mathbin{⟖} R$) Functionally the same as the left outer join, but with the sides reversed.

Full outer join ($L \mathbin{⟗} R$) Effectively a combination of a left outer join and a full outer join. The result set consists of all join matches between L and R, as well as all unmatched records from L and R (where the columns of the unmatched are populated using `NULL`s). The result set therefore contains at least $\max(|L|, |R|)$ records.

Semi-join ($L \ltimes R$) A join between L and R, but with only the columns of L as output. Unmatched records are not part of the result set. Can be interpreted as a projection of $L \bowtie R$.

Anti-join ($L \triangleright R$) Outputs all records of L for which – in a hypothetical join – no matching record exists in R. Strictly speaking, this is not a join in the original sense, but its functionality is closely related to that of other join types. Since the result set of an anti-join is equivalent to every record of $L - (L \ltimes R)$, it is also called an 'anti semi-join'.

Cross product ($L \times R$) An indiscriminate join of L and R, such that the result set is the product of L's size and R's size. Also called a 'cross join' [9].

A simple SQL example for some of these join types is depicted in Subsection B.2.4.

It is possible to use more than two tables in a query, though internally a join is generally between two result sets. If more than two tables need to be joined using binary join operators, this is generally done by composing a join tree in which each leaf corresponds to one table. When queries have arbitrarily many tables to be joined using join operators, it becomes a non-trivial task to determine which join ordering (tree of joins) is appropriate. In fact, the finding the optimal join ordering is an NP-hard

problem, meaning that it is not viable to determine the absolute optimal join sequence for non-trivial amount of joins [10].
For the scope of this thesis, it is assumed that joins are always binary, though non-binary join operators (multi-way joins) do exist [11].

System-specific join types

Some RDBMSs contain further join types. Namely, the HyPer RDBMS internally also distinguishes a *dependent join*, *single join*, and *mark join*, each of which are introduced for optimisation purposes when queries contain (complex) subqueries [9]. These particular joins are special cases of existing joins, which – if treated as a special join type – can be executed much more efficiently than their 'generic' counterpart.

A *dependent join* matches one record on its left-hand side (LHS) to an aggregation of (eligible) records on its right-hand side (RHS), meaning that the output column on the LHS is the result of an aggregate (sub)query. A *single join* matches one record on its LHS to at most one record on its RHS, where a non-match results in a `NULL` (and multiple matches in an error). The RHS of a single join represents a scalar, and must therefore consist of exactly one column. Finally, a *mark join* checks the existence of a join partner. For each record on the LHS, a mark join yields `TRUE` if the RHS has at least one match for the join condition of interest, and `FALSE` otherwise. This join type is necessary for performance reasons if, for example, this boolean value is needed in conjunction with some other expression.[9]

2.5.2. Runtime complexity

Join conditions usually evaluate whether some identifier on the LHS is equal to another identifier on the RHS, but join conditions can be arbitrary. Although semantically dubious, a join condition could for example evaluate whether a `revenue` column on the LHS is larger than the expression $\sqrt{\texttt{age}}$ on the RHS (using its `age` column). For arbitrary expressions, it may be necessary to compare every individual record on the LHS of a join to every individual record on the RHS of a join. Using L, R, $|L|$, $|R|$ as defined in Subsection 2.5.1, an arbitrary join may thus have a complexity of $O(|L| \cdot |R|)$. A join that is executed in this manner is called a *nested loop join* (or *naïve join*). This is however a worst case, and in practice there are almost always optimisations possible that do not yield such a complexity.

To this end, it is important to first understand what data two different tables can have in common, since join conditions are usually formed thusly. Tables may have a formalised relationship with each other through the usage of identifiers (IDs). Many

[9] According to [9], it is still slightly faster to use an anti-join if the query allows for this.

database systems support a *foreign key* constraint, which stipulates that an identifier – or set of identifiers – used in one table must be defined in some other (specified) table. For a foreign key constraint to be possible, the other party of the relationship must have a *unique* constraint on this key; this means that only one record in that table may use a specific key. Defining constraints (including keys) has its trade-offs:

- Constraints may guard the correctness of the data inserted into the database, at the cost of additional checks when inserting or modifying data.
- To support constraints, it is often necessary to introduce an index. As discussed in Subsection 2.2.3, index can drastically improve performance when needing to access specific records, but maintaining an index can negatively affect performance when regular insertions take place.

Identifiers are commonly used inside join conditions – whether a foreign key relationship is defined or not. Identifiers are most commonly some form of integer, or at least a data type with some lexicographical ordering defined. When this is the case – which is almost always – it is possible to *sort* the output of both join sides before evaluating join conditions on records between the tables. Such a join is called a *sort merge join*. When the predicate is an equality operator, it also becomes possible to *hash* both join sides, yielding a *hash join*.

Sorting and/or hashing greatly reduce the complexity of performing joins. Since the vast majority of joins can use either of those strategies, it is rare – though not always avoidable – for a nested loop join to be needed when performing a join.

To conclude, a join theoretically has a worst-case complexity of $O(|L| \cdot |R|)$, but in practice most joins are executed orders of magnitude faster than this worst case. Optimisations for join types that are not explicitly represented in SQL can further improve query performance.

2.6. Query execution in databases

As mentioned previously, users need to write SQL queries to read, insert, or modify data in the database. These queries are declarative, meaning that it is up to the database system how to execute the user query.

Internally, a database system converts a SQL query into a query execution plan (often named a logical plan). This logical plan is represented by a tree of operators, to be executed bottom-up, using a traversal algorithm such as Depth-First Search (DFS). Most operators in a logical plan can be represented by symbols in RA. Some operations are database-specific, and do therefore not have a conventional algebraic representation.

Roughly speaking, each main keyword of the SQL language is represented by an operator in RA. Indeed, SQL is a practical implementation of Codd's RA originating from the early 1970s [2], [12]. RA can be rewritten to SQL and vice versa, and thus the two are effectively a different notation for the same underlying operations. For this section, it is assumed that the reader is aware of the most common RA symbols, which are also depicted in Appendix A for reference.

The explanation in this section is twofold: Subsection 2.6.1 describes how RA can be used to represent a query in a tree structure. Finally, Subsection 2.6.2 shows how this tree can be improved, with an optimised logical plan as result.

2.6.1. Logical plan creation

The initial state of a logical plan is in essence a naïve conversion of a user's query to a tree structure. The exact procedure for this may depend on the RDBMS, but is generally a direct conversion from the individual clauses in a SQL query to their respective operators in RA, with some special handling for subqueries. Using the query in Listing 2.3 as an example, a naïve conversion to a logical plan may look like the tree in Figure 2.1.

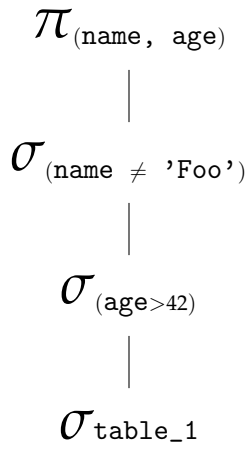


Figure 2.1.: Naïve (unoptimised) logical plan of the query in Listing 2.3.

Operators in the logical plan generally have one or two children. Most operators have one child, whereas set operations and joins have two children. Notably, a selection (σ) may be childless, making it a leaf in the tree. This occurs when the operator is used to retrieve a base relation (as can be seen in Figure 2.1).

The query tree is to be interpreted as a pipeline flowing bottom-up. As the name sug-

gests, an operator performs a specific *operation* on its input, the result of which it pipes to its parent operator. The root operator, having no parent, either outputs the result to the user, or writes any changes to the RDBMS (in the case of an insert/update/delete). The query plan in Figure 2.1 therefore runs the following steps:

1. Obtain the data from `table_1`,
2. Filter the records with `age` above 42,
3. Exclude any records where `name` matches 'Foo',
4. Create a projection using the following columns: `name, age` (in that order).

Each operator is its own step, and each operator handles their step isolated from the other steps.

2.6.2. Optimisation

It is trivial to see that the procedure depicted by Figure 2.1 is not the most efficient method to execute the query in Listing 2.3. There are three selection operators, which means that the data of the base table is needlessly scanned repeatedly. A common method to optimise queries is to perform a *predicate pushdown*. This means that any selections that filter records in the result set should be executed as early as possible in the query tree. Doing so successfully means that any remaining operators need to process fewer records.

Using predicate pushdown, Figure 2.2 depicts an improved version of the plan in Listing 2.1. Programmatically, the most notable difference is that only one selection is performed, meaning that the base table records are only iterated over once. In the first plan, all queries would first be fetched, then filtered by age, and only then filtered by name. In this optimised plan, the fetch of the base table already filters on both the name and age record by record, such that each record only needs to be processed once.

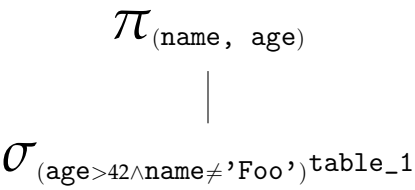


Figure 2.2.: Optimised logical plan for the query in Listing 2.3, using a simple predicate pushdown.

Similarly, many database systems tend to push projections down the tree. The rationale for this is simple: projections generally result in fewer columns, and fewer columns early on results in less data being passed between operators. Naturally, less data means that the remaining operators have less to process, and are thus more efficient. Although this is not needed in this simple example, any joins in a query plan can greatly benefit from dimension reductions, wherever they are possible.
Predicate pushdown and projection pushdown may change the order in which actions are executed thusly. In addition, joins and subqueries may radically change the structure of the logical plan during optimisation. With arbitrarily complex queries, the resulting optimised query may therefore look significantly different to the structure of its defining SQL query.

3. Incremental View Maintenance (IVM)

To properly understand the scope of this thesis, it is important to understand what IVM is about, and how it relates to database management systems as a whole.

This chapter describes IVM, as well as some of its implementations. First, Section 3.1 describes the notion of IVM, after which Section 3.2 denotes some current IVM systems (as well as their characteristics). Following that, Section 3.3 then describes OpenIVM – the IVM system to which this thesis contributes.

3.1. What is IVM?

IVM extends upon the view logic described in Section 2.4. Traditionally, the aforementioned materialisation techniques query all relevant data from the base tables every time a view has to be updated. This means that, in some cases, even a single update can cause the entire view to be recomputed, in particular when aggregates such as sums and averages are included in the view.

Rather than recreating the view in full for every change, IVM strives to update the view *incrementally*. This means that only those rows that have to be inserted, updated or deleted are actually modified, while keeping the rest of the materialisation untouched. In most cases, this should greatly reduce the load on the database when it comes to maintaining views. However, it does mean that a database system has to perform more bookkeeping to keep the view data accurate. Bookkeeping is done in some sort of differential log, which for instance can be done by using a separate table to store changes in. Alternatively, one could also scan the same sources using versioning or a timestamp, which would subsequently necessitate the use of a means to identify each row uniquely (so-called row ID, for merge operations). Especially when a view requires data from several tables, the bookkeeping (and subsequently, figuring out what changes have to be made to the view materialisation) can become non-trivial.

3.1.1. Parallels with stream processing

Stream processing for a long time has been a research field relatively isolated from RDBMS development. However, in recent years, the two research fields have been converging, with many systems now using SQL as their interface [8].

The concept of IVM may also be viewed from a different perspective, namely that of stream processing. A stream of data means that the data is unbounded in dimensions, meaning that it is impossible to materialise any results until the data is complete.

IVM can act as a bridge between stream processing workloads and more traditional RDBMSs. Rather than collecting data, loading it in an OLAP database once, and only then drawing conclusions from the data, stream processing makes it possible to directly analyse data 'from the source'. If RDBMSs can properly support IVM, this can in turn greatly benefit the performance of stream processing workloads on such systems.

3.1.2. Limitations

Although IVM should be able to support most queries to a database, not every database operation is reasonable to use IVM for. This is mainly the case for certain aggregation operators like `min`, `max`, or `median`.

To illustrate this notion, suppose an arbitrary base table with 2000 records is used for an MV yielding a scalar aggregate. If 5 records are then removed from the base table, and the MV uses an aggregate such as `sum`, the updated sum can be computed by subtracting the sum of these 5 new values from the old sum. However, when the used aggregate is instead a `median`, it is not possible to determine the new aggregate using exclusively the old aggregate and the deleted records.[1]

Specifically for joins, the challenges for implementing joins and specific join variants are discussed in both Chapter 5 and Chapter 8.

3.2. Related systems

As of present, a number of IVM implementations have already been developed. Most of these systems use a dedicated engine for the previously mentioned bookkeeping, rather than incorporating it into the engine of an already existing database system. As a consequence, many of these implementations remain mostly theoretical, and currently have no practical usage. Others may be solutions to a specific proprietary product, but without providing support to an RDBMS.

[1] Rarely, it can be determined with certainty that the median does not change, and therefore no recalculation would be needed: namely, when modifications, insertions, or deletions take place in such a manner that the number of values above the median (or value pair forming the median, depending on parity) remains equal to the number of values below the median value (or value pair). However, since this is merely keeping a view 'intact' and not maintaining it, this 'optimisation' on its own does not suffice for IVM.

3.2.1. DBToaster

One example of a system using its own engine is DBToaster, which makes intensive use of eager materialisation [13]. Eager materialisation means that – to maintain the view – this system also materialises intermediate results, without the end user explicitly requesting so.

DBToaster is especially relevant in the context of joins. The reason for this is its approach for computing the delta (difference) on a join. The approach described in DBToaster is to compute the difference made to a join using a union of three joins.

In terms of complexity, to perform IVM while avoiding linear (or worse) complexity on the large join input, one needs to have that table materialised with an index on the join key. In that case, a "nested-loops index-lookup join" can be used – with the delta as the outer relation in this nested join – in which each nested lookup has logarithmic complexity on the large join input. An index requirement is functional only if the join input is a table. If the input is *not* a table – as is the case for subqueries and other more complex query plans – this cannot be used.

DBToaster was the first IVM approach to radically materialise all query tree nodes which are join inputs inside an IVM – if these join inputs are not already base tables. To maintain the top-level MV, the query tree may thus see auxiliary MVs being made recursively – meaning that the maintenance of one MV may result in an arbitrary amount of additional MVs. Each of these individual MVs still need to be maintained; given the recursive nature of the query tree, this maintenance has to be done in a bottom-up fashion. This approach therefore trades off the significant cost of additional materialisation with avoiding the complexity of the IVM (which is linear to the base tables involved in the MV).

3.2.2. DBSP

Another system based around a dedicated engine is DBSP [14]. The concepts presented by DBSP are also applied in OpenIVM [1]. DBSP describes itself as a 'language' to describe (incremental) computations on data streams. The name *DBSP* is a combination of DB (abbreviation of 'database') and DSP (digital signal processing). As the name may suggest, DBSP interprets data as a stream that can be processed similarly to signals. Operators are represented as gates. Instead of query trees, DBSP represents queries as circuits. Signals (in this case, the data) are fed through the circuit, and the output of the query is equivalent to the output of the circuit. An implementation of DBSP is in active development.[2]

[2] `https://github.com/feldera/feldera`

DBSP should theoretically support most of the SQL language, hence constituting a functional approach to IVM. As a notable exception, some generally hard-to-materialise aggregates such as `min`, `max`, or `median` are not possible to incrementalise.

3.2.3. Dynamic Tables in Snowflake

As mentioned in Subsection 3.1.1, IVM can be considered from the perspective of stream processing. This is exactly the approach of Snowflake's Dynamic Tables [8].

Dynamic Tables are Snowflake's version of IVM, using SQL as the interface to obtaining insights from stream data. The core notion behind Snowflake's implementation is the concept of *delayed view semantics*. The term 'delayed view semantics' refers to something that was briefly discussed in Section 2.4: namely, an MV should always represent the state of the base tables either at present or at some point in the (recent) past. The Dynamic Tables feature is built on top of Snowflake's existing functionality, and is used in production.

3.3. OpenIVM

Instead of writing a dedicated engine for IVM operations, OpenIVM strives to define all operations using SQL queries [1]. OpenIVM uses the DuckDB RDBMS for implementing all IVM functionality.

The motivation behind OpenIVM is multifaceted. Since the IVM operations can use a database system's query engine, this particular implementation of IVM should be a lot easier to support in practice. Not only does it save the effort of having to maintain multiple separate engines, it also makes the IVM portable across systems. OpenIVM can be adopted by any database system that supports SQL. Finally, by using the database system's native query engine, OpenIVM can also automatically benefit from its query optimiser. This is of great use to process updates in an efficient manner. These aspects together pave the way for 'cross-system IVM' [1] for Hybrid Transactional/Analytical Processing (HTAP) systems. As the name suggests, HTAP is a hybrid combination of OLAP and OLTP, which – using OpenIVM – enables pipelines across RDBMSs for IVM purposes.

OpenIVM is a SQL-to-SQL system, and all materialised view creations and maintenance are done by creating bespoke SQL queries for them. OpenIVM's differential log is implemented using so-called *delta tables* – denoted by a Δ in the remainder of this document. These tables indicate the changes made to the relevant base tables since the last time the materialised view has been updated, such that incremental changes to the

view become possible. The delta table contains the same columns as its respective base table, but with two additional columns:

Multiplicity column The delta table must have a boolean *multiplicity* column to determine whether a change is an addition (`true`) or a deletion (`false`). Internally, this column is named `_duckdb_ivm_multiplicity`.

Timestamp column The delta table must also include some sort of indication on *when* a certain record is inserted, added, or modified. This is done through an additional *timestamp* column (internally: `_duckdb_ivm_timestamp`).

For IVM, the original query for the MV gets modified to use the delta tables, such that any changes can be propagated into the MV. The complete OpenIVM procedure for converting an MV declaration to an IVM declaration is as follows:

1. A user declares an MV query using SQL.
2. The query gets converted into a logical plan, which is then optimised by the DuckDB optimiser.
3. The logical plan gets modified recursively – bottom-up – to use (the data of) the delta tables instead.
4. This modified logical plan is then converted into SQL, along with some further queries to modify data in the base tables of interest.

The logical plan and its rewrite are further elaborated upon in Subsection 3.3.1. The conversion of this modified plan into SQL is then covered in Subsection 3.3.2. Finally, Subsection 3.3.3 discusses some implementation details of aforementioned logic, including details on the timestamp and multiplicity column.

3.3.1. Rewriting the logical plan

The interface of OpenIVM is SQL, and therefore the unoptimised query is a result of the user's input converted into operators. Such queries can be arbitrarily complex. Thus it is beneficial to optimise the query insofar as possible before any IVM-specific changes are made, since this may simplify the query that OpenIVM has to perform its modifications on. To this end, OpenIVM uses the DuckDB optimiser.

Besides modifying the plan for the materialised view itself, the OpenIVM code also makes adaptions to queries that modify (insert, update, delete) any base tables of interest. Even read-only queries to the base table (without using any MVs) have to be

corrected, such that any potential records that are currently in the delta table are also properly propagated for other queries using the base table.

The changes needed for composing the modified logical plan in OpenIVM are best explained with an example. Let R be a base table, and $V(R)$ be some materialised view using R. ΔR is thus the differential table for R. Any updates made to $V(R)$ – whether insertions, updates, or deletions – concern records inside ΔR. Thus, the declarative query for $V(R)$ – in particular its logical plan – can be modified to generate OpenIVM's maintenance query. The idea behind this is simple: inside the logical plan, replace the base table R with ΔR and adapt the remaining operators in the logical plan. However, further modifications are necessary, since the additional multiplicity column of ΔR needs to be passed throughout the tree, and thus all operators should be adapted accordingly. This adaption, as well as the handling of ΔR's timestamp column, are described in Subsection 3.3.3.

Beyond the changes that are directly related to the multiplicity column, the computational changes that are needed for many operators are in itself non-trivial as well.

The result of this rewritten query – having adapted both the base tables and modified the computational tasks of operators where applicable – is inserted into $\Delta V(R)$, which consists of the columns as $V(R)$, and additionally, a multiplicity column that is congruent to ΔR's multiplicity column. Finally, the records of $\Delta V(R)$ are propagated into $V(R)$ in a manner appropriate to the original query. For instance, if the original query contains an aggregation and $\Delta V(R)$ yields an addition relevant to a particular aggregate, this should result in an *update* of the respective record(s) in $V(R)$, and explicitly **not** a new record separate to the existing record.

Upon completion of these steps, the 'last updated' timestamp of $V(R)$ should be updated within OpenIVM's *system tables*, which comprise the operational metadata of OpenIVM.

3.3.2. Compiling SQL queries

The SQL query is then executed by the respective database systems. As stated previously in Chapter 2, this query merely has to semantically convey the intention of the query – within the syntax rules of SQL – after which it is up to the optimiser of those other systems to run the query efficiently. This is the quintessential thought behind OpenIVM.

Unlike most human-written SQL queries, the output of the SQL-to-SQL query generator may not necessarily be easy to read for humans. After all, these queries are not meant to be read (nor created) by the end user, but instead used to instruct an RDBMS which updates need to be made for IVM.

The main hurdle for cross-compatibility across SQL systems is the existence of various dialects in SQL. For some RDBMS systems, the query created by OpenIVM may have to undergo some syntactical changes before they can be used in these systems.

The implementation of the SQL compiler for OpenIVM is discussed further in Chapter 6.

3.3.3. Implementation details

As mentioned in Subsection 2.3.3, data can be modified in SQL using inserts, updates, and deletes. In IVM terms, there are positive and negative changes. Inserts correspond to positive changes, and deletes correspond to negative changes. An update is represented using a combination of a deletion and an insertion.

Since all changes are to be stored in the delta table, it is necessary to store whether a change is an addition or a removal. As mentioned previously, this is handled by the multiplicity column.

The MV has its own delta table containing the records that are about to be propagated into the MV. This delta table includes a multiplicity column, but does not include a timestamp column.

Recall that the logical plan is modified recursively, in a bottom-up approach. Multiplicity columns should therefore be passed through from all operators in a query tree – from the leaves containing them up to the root of the logical plan. By convention, the multiplicity column – if present – should always be the final column of an operator. Although for most operators this is merely a matter of adding an additional column, special care needs to be taken for specific operators:

Aggregations (Γ) need to distinguish between positive and negative changes, and thus must add the multiplicity column to its `GROUP BY` columns.

Set operations ($\cup$, $\cap$, $-$) require the columns of its LHS and RHS to be aligned. The multiplicity column may potentially cause problems to this end, which have to be accounted for.

Joins ($\bowtie$) need special care depending on whether *either* or *both* sides of the join contain a multiplicity column. If only one of the sides contains a multiplicity column, then a projection should be added to ensure that the multiplicity column remains the final column. If both sides contain a multiplicity column, an additional join condition is needed to exclude records whose multiplicity column are incongruent. This is elaborated upon in Section 5.3.

The delta tables for base tables also contain a timestamp column. As mentioned previously, this column indicates when this record was added to the delta table. Inside

the logical plan rewrite, these timestamps are only used to filter records on leaf nodes. The timestamps are therefore not passed through to any other operators in the plan.

One might think that – instead of using timestamps – it would also be possible to wipe the delta table upon each update, meaning that a timestamp-based selection on the delta tables would no longer be necessary. However, having a timestamp column is useful for several reasons.

First, the timestamp column makes it possible to use the delta table across different MVs. If two different MVs are made that share one or more base tables, then there is no need to main multiple differential tables per base table. Instead, each MV stores its last update time and can then compare that timestamp to all recorded changes in the delta table. Only those changes since the MV was last updated are then propagated into the IVM procedure. The ability to reuse delta tables for multiple MVs avoids unnecessary overhead during insertions. Second, if each record in the delta table has an annotated insertion timestamp, the delta table can function as a log of changes in that table.

4. DuckDB

DuckDB is an open-source column-oriented OLAP database system [15]. In its foundational paper, Raasveldt and Mühleisen [15] demonstrate DuckDB as the OLAP equivalent of SQLite. Whereas SQLite is an embedded OLTP system (in contrast to many stand-alone OLTP systems), the 'landscape' of OLAP systems did not appear to have any embedded systems at the time of DuckDB's creation. By being an embedded OLAP system, DuckDB can be used in workloads that would otherwise use Python packages such as Pandas, but also other data science applications or data pipelines [1], [15]. DuckDB is written in C++, and supports the creation of extension modules to extend its functionality [1], [15]. Notably, OpenIVM is implemented as an extension for DuckDB.

This chapter covers the characteristics of DuckDB, given its relevance for OpenIVM. Section 4.1 contains a quick overview of DuckDB's features and structure, highlighting what is different to most other RDBMSs. Then, Section 4.2 covers internal aspects of DuckDB insofar relevant to this thesis. With this knowledge at hand, the remaining chapters can more precisely refer to specific properties of DuckDB.

4.1. Characteristics and feature support

In this section, the characteristics of DuckDB are described, such as how data is stored and which design decisions were made in DuckDB regarding features and execution.

As an OLAP system, DuckDB makes use of columnar storage, which is combined with vectorised execution to extract high performance. However, DuckDB has limited support for indices compared to most other database systems. Defining primary keys and foreign keys almost never yields any performance benefits (and may even hamper performance). This has consequences when wanting to explicitly define/declare relationships between tables, but should not affect performance of most OLAP-like queries. DuckDB, however, does support zonemaps and Adaptive Radix Tree (ART) indices – both of which are helpful for selective operations on a specific table. However, both zonemaps and ART indices do not affect the performance of joins, aggregations, and sorting, and DuckDB does not recommend the use of keys if they are not absolutely

necessary to enforce constraints.[1]

As mentioned in the introduction of this chapter, DuckDB is an *embedded* RDBMS. This means that DuckDB is meant for local usage (stored on a user's device, and exclusively for this user). This is thus the opposite of a large (distributed) system hosted on a dedicated server, which adds latency overhead for each individual query – on top of significant setup effort required before first usage.

DuckDB uses the same SQL parser as the PostgreSQL RDBMS [1]. This means that any SQL query written for one of those systems is highly likely to be compatible with the other system. This is not necessarily the case for queries written in other SQL dialects.

The DuckDB optimiser also supports the extended join types of the HyPer RDBMS, which were mentioned in Subsection 2.5.1. DuckDB currently does not have support for MVs, though the in-development OpenIVM extension is effectively an implementation of MVs for DuckDB.

4.2. Internal interface

Like virtually all RDBMSs, DuckDB uses a tree-based query logical plan to execute queries. In this section, the execution of such a logical plan is explained, such that it becomes clear how the individual operators in a logical plan pass data to each other and ensure that the query plan is logically sound.

This is best explained by looking at leaf nodes. Each leaf of a logical plan is some sort of *scan*. Most commonly, this is a scan of a base table (named a `LogicalGet` in DuckDB), but scans from external files, constants declared inside a query, or even empty result sets can also be a leaf node. Leaf nodes are important, as they are the only nodes that obtain data from outside the logical plan. This explanation mostly elaborates upon `LogicalGet` operators, but for all intents and purposes it can be assumed that the other leaf nodes have an equivalent logic.

4.2.1. Representation of tables and columns

In DuckDB, each logical operator – including a `LogicalGet` – has *Column Bindings*. A Column Binding (CB) is a pair of a *table index* and *column index*, and uniquely identifies a column within the query plan. Both table indices and column indices are zero-based.

CBs are defined within operators that generate new table indices. Inside DuckDB, these are the following operators:

- A `LogicalGet`, as well as any other leaf node.

[1] `https://duckdb.org/docs/stable/guides/performance/indexing.html`

- A `LogicalProjection` (equivalent to a π in RA).
- Any `LogicalSetOperation`, notably a union ($\cup$ in RA).
- A `LogicalAggregate` (equivalent to a Γ in RA), which generates three different table indices, each for another purpose. This is the only operator that generates more than one table index.

All these operators have in common that any CBs cannot be used without potentially introducing ambiguities. The reasons for this are diverse. A `LogicalGet` has no children, so it must define its own bindings. A `LogicalProjection` may duplicate columns, cast them to another type, generate new columns based on constants, or concatenate data from several columns into a new column. The output of (each column of) a `LogicalSetOperation` contains input from two different children – and hence, different column bindings – meaning it would be semantically incorrect to inherit the CBs of either child. Finally, a `LogicalAggregate` defines new columns that are a result of one or more aggregate operations.

It must be noted that the set of operators that generate new table indices is not identical to the set operators that materialise their output. An important counterexample is the join operator: in DuckDB, a join inherits the CBs of its children, but often does require a materialisation when executing the query (for example, for a hash join).

CBs are used to *bind* columns inside the logical plan to physical database tables. This is done by the *binder* inside the database. After the binder completes its execution, all CBs are accompanied by their type inside what is named a *bound column reference*.

An explicit linking between an operator and a base table takes place within the `LogicalGet` operator. A `LogicalGet` defines its CBs using a vector of Column IDs. Column IDs refer to the identifiers that columns have within a base table, and are expressly different from the CBs that any operator in the logical plan uses. Figure 4.1 depicts this difference using an example, and shows how CBs are derived by the `LogicalGet` operator using a base table.

Figure 4.1a depicts the base table, which in this case has table index 0. Figure 4.1b depicts the vector of ColumnIDs that the operator wishes to obtain, in that specific order. Figure 4.1c shows the resulting CBs, annotated with the names of the base table columns that they correspond to.

In this example, it can be observed that CBs are assigned in increasing order, from left to right by the operator that declares them. There is no guarantee that this order of CBs stays intact inside any parent operators.
For non-leaf operators, CBs are generally defined using *expressions*. These expressions instruct how a column should be derived from, for example, the child operators. This is explained further in Subsection 4.2.2.

Sales
order_id
p_name
amount
date_ordered

(a) Base table.

column_ids
[1, 2, 0]

(b) Vector of Column IDs.

p_name	amount	order_id
[0,0]	[0,1]	[0,2]

(c) Resulting CBs.

Figure 4.1.: Example conversion of a base table into CBs. On the left is the base table (with table index 0). The middle shows a vector of Column IDs as defined by the `LogicalGet`. The resulting CBs are depicted on the right.

In short, CBs indicate a distinct column introduced by a table-index-generating operator. When bound, they are accompanied by their type inside a bound column reference.

4.2.2. Data flow between operators

CBs are instrumental to pass data from one operator to another. Whenever an operator does not generate new CBs (i.e. any operator not mentioned in Subsection 4.2.1) it inherits columns from its ancestors. For example, a `LogicalFilter` performs a selection of records based on provided conditions (in the form of expressions). Since the columns themselves are not modified, neither are the CBs; they are merely inherited from the child operator and data can be unambiguously transferred thusly.

This is particularly important to understand for joins. Inside DuckDB, joins do not generate a new table index, but instead inherit the CBs from both of its children. This consequently means that CBs in a join can be derived from arbitrarily many table indices, since joins can have arbitrarily many joins as their ancestors. Join conditions in DuckDB function logically similar to the explanation in Section 2.5.

4.2.3. Plan verification

After DuckDB has created an initial plan, has bound all CBs, and has optimised the query, the database system runs a verification process on the logical plan. This verification process (henceforth called the `Verify` function) ensures that the query is

sound, all CBs are not ill-defined, and that any used table indices can be mapped to a specific operator.

More specifically, the `Verify` function checks – for each operator with two children – whether any table indices used by descendants are used by only one of those descendants. Therefore, If any table index is used on both sides of a binary operator, this is illegal and will yield a failure in the `Verify` function.

Also checked is whether any specific CB (of a non-CB-generating operator) has been defined by the direct child (or children) of the operator. For example, if an operator defines CBs `[7.0]` to `[7.4]`, then a CB of `[7.5]` appearing in its parent operator would imply a structural error in the plan, yielding a failure in the `Verify` function.

Part II.

Approach

5. Joins in OpenIVM

This chapter covers the implementation of joins in OpenIVM. For this implementation, the focus has been specifically on inner joins, since it is the most common join type in RDBMSs.

First, Section 5.1 will devise the theory behind joins in OpenIVM. Subsequently, Section 5.2 shows that this recursive logical plan modification is logically sound for a query with multiple joins. In Section 5.3, it is discussed whether new join conditions need to be introduced. Finally, Section 5.4 covers the practical steps required to implement joins in OpenIVM, as well as complications that arose in this implementation and were resolved.

The discussion in Chapter 8 will elaborate on the requirements and challenges to implement other join types in OpenIVM.

5.1. Theoretical explanation

Recall from Section 2.5 that a join combines an LHS and RHS based on one or more join conditions. This section describes how a logical plan containing joins – represented as a tree in RA – should be modified for the IVM maintenance step.
Consider the query tree structured as in Figure 5.1. Here, L and R may be arbitrary subtrees of any size. For simplicity, the remainder of this section refers to L and R (and their respective differential outputs) as *tables*. The changes in L and R are stored in so-called *delta tables*, which are only emptied when the view is updated.

For IVM, the join would have to be decomposed into three separate joins, whose result set will be combined using `UNION`s. The reason for this is that L, R, and any changes made to L and/or R can yield three different combinations in which the result of the view changes:

1. A change in L (represented as ΔL) may match with any existing records in R.
2. A change in R (represented as ΔR) may match with any existing records in L.
3. A change in L (ΔL) may match with any changes to R (ΔR).

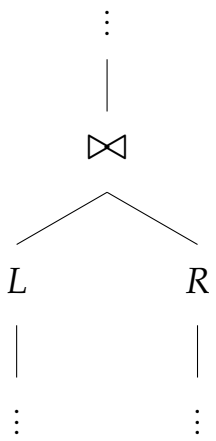


Figure 5.1.: An example query tree with a join ($\bowtie$) between subtrees L and R.

The modified plan should therefore contain the following three joins: $\Delta L \bowtie R$, $L \bowtie \Delta R$, and $\Delta L \bowtie \Delta R$. Notably, any existing records of L and R need not be joined into $L \bowtie R$, since this is the current state of the view. Accounting for those three joins, a version of the tree in Figure 5.1 modified for IVM is depicted in Figure 5.2.

Recall from Subsection 3.3.3 that – in contrast to the original query's tables – each delta table includes an additional multiplicity column. Although the logical tree in Figure 5.2 could theoretically work, in practice there is a further modification required for this query to function. Namely, for a `UNION` to work in a relational database system, both children of the union must have their columns arranged in an identical manner. Consequently, compared to the original query, the following two issues arise:

- The original query did not have a multiplicity column. In this query tree, two joins include a singular multiplicity column in their output, whereas one join includes two multiplicity columns in their output.
- The original query may have a projection after the join, since joins generally do not handle projections themselves. Since a union mandates an identical arrangement of columns for both children, any such projections may also have to be included as children of the union.

Thus, the following changes have to be made to the query tree. First, a projection should be inserted above each join, such that this projection becomes a child of a union. Second, each operator in the tree should be modified to either include or handle a multiplicity column correctly.

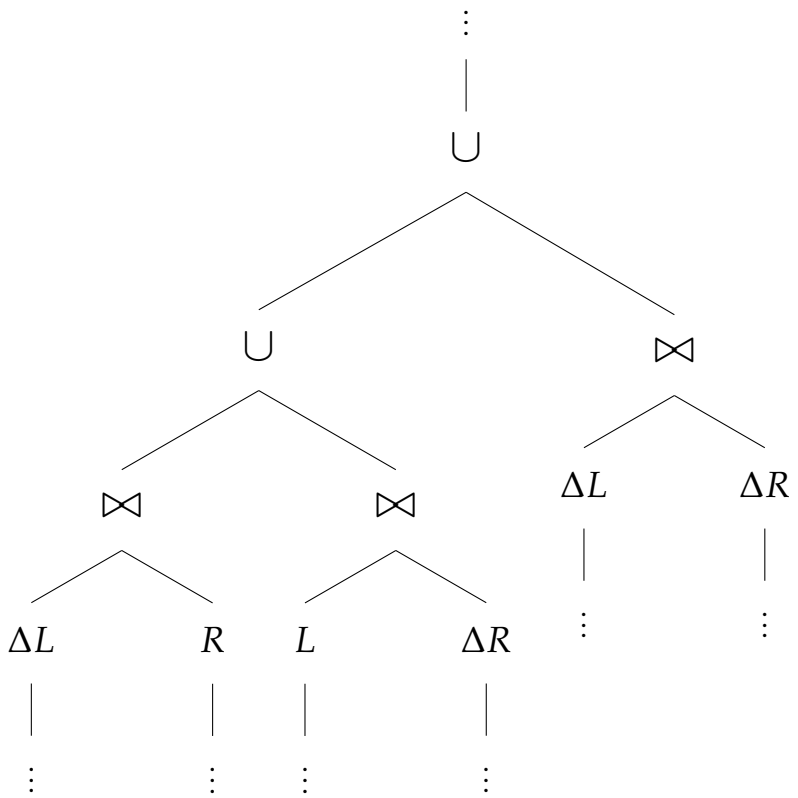


Figure 5.2.: IVM representation of a tree with a join.

Lastly, as briefly mentioned in Subsection 3.3.3, the join $\Delta L \bowtie \Delta R$ requires an additional join condition to handle the two multiplicity columns. This is further elaborated upon in Section 5.3.
When adapting the query tree to the above, the resulting query tree would look like in Figure 5.3.

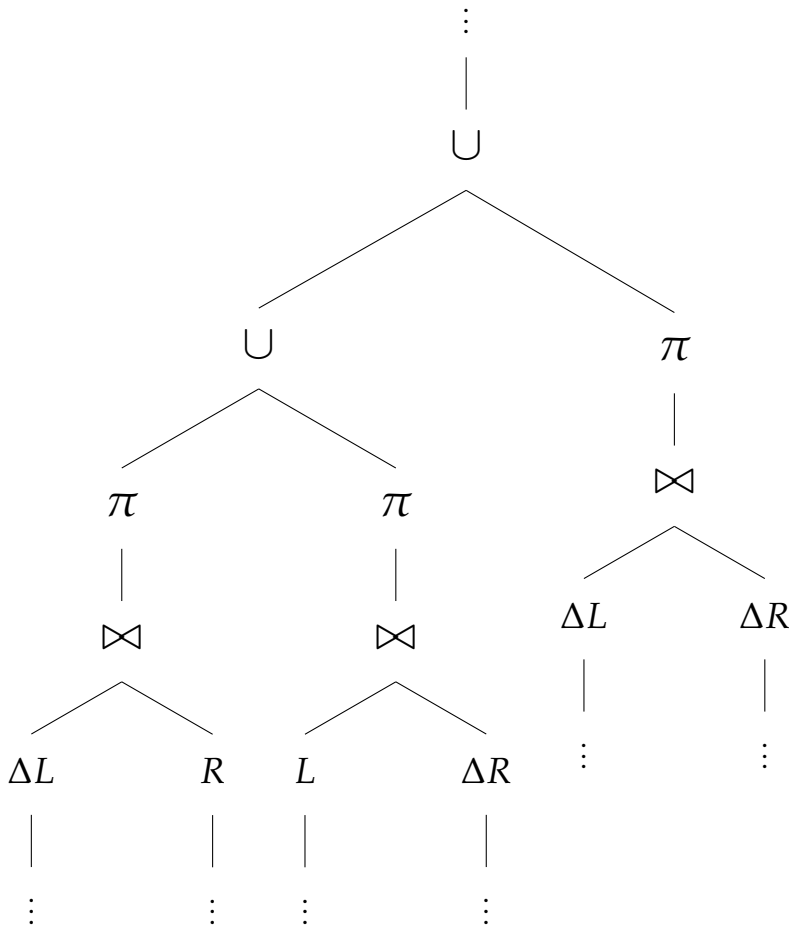


Figure 5.3.: IVM representation of a tree with a join, with additional projections beneath both union operators to ensure compatibility.

5.2. Query plans with multiple joins

In this section, the recursive nature of join rewrites is explained, to show that this logic indeed functions with multiple joins.

Consider the query tree in Figure 5.4, depicting $(A \bowtie B) \bowtie C$ with arbitrary subtrees. Since the query is bottom-up recursive, $A \bowtie B$ will be rewritten first. This rewrite is done in accordance with the explanation in Section 5.1, and thus makes the (partially) rewritten tree look as in Figure 5.5.

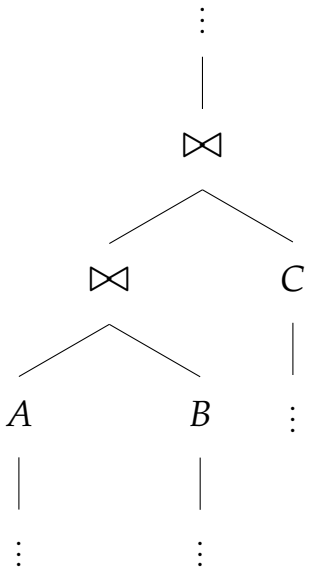


Figure 5.4.: Query tree with multiple joins, in this case between A, B, and C.

The LHS of the topmost join in Figure 5.5 now corresponds to the output of $(\Delta A \bowtie B) \cup (A \bowtie \Delta B) \cup (\Delta A \bowtie \Delta B)$ after any necessary projections, which shall henceforth be referred to as $\Delta(A \bowtie B)$. The main question now becomes what C needs to be joined with such that the logical plan yields the correct output for IVM.

To show that the query tree can indeed be modified recursively for base query containing multiple joins, let the join $A \bowtie B$ in the aforementioned equations be substituted by the variable L. When C is subsequently substituted by R, the join pairs to include in the IVM are identical to the ones in Section 5.1, showing that this recursion is indeed a functional approach to supporting multiple joins in one MV query for IVM. The result set of the modified tree should therefore contain $\Delta(A \bowtie B) \bowtie C$, $(A \bowtie B) \bowtie \Delta C$, and $\Delta(A \bowtie B) \bowtie \Delta C$. The resulting query plan is shown in Figure 5.6.

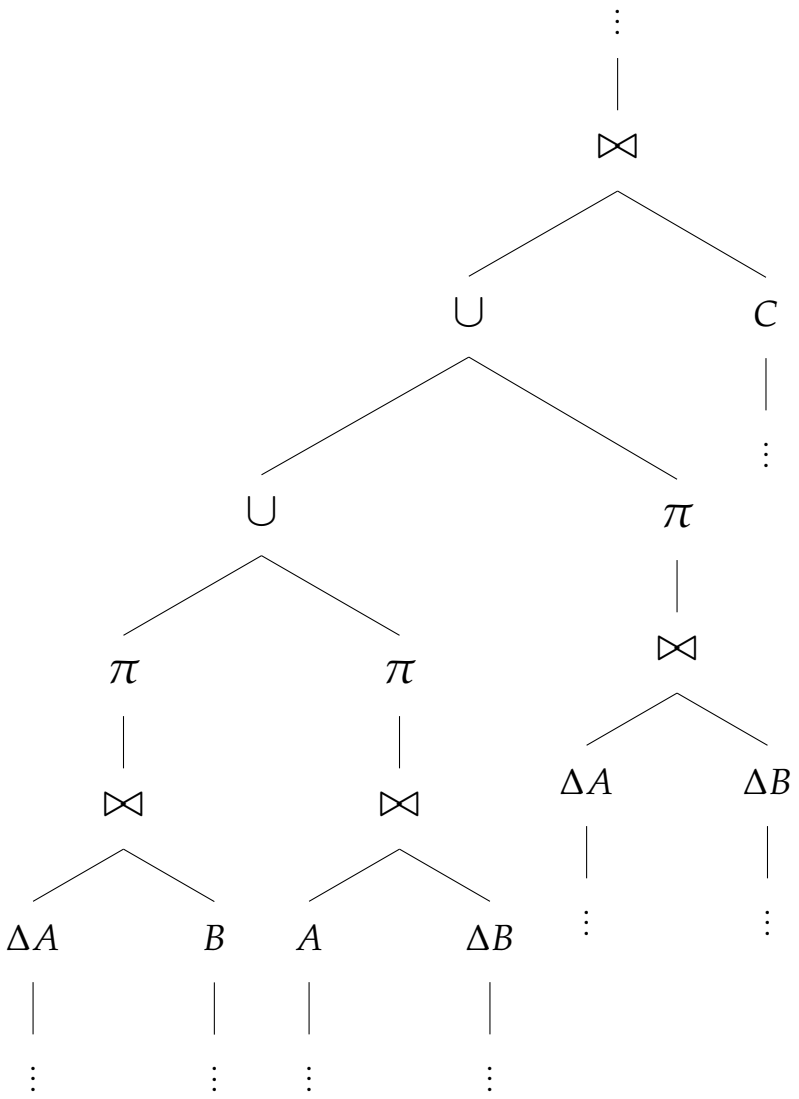


Figure 5.5.: Partially modified query tree for $(A \bowtie B) \bowtie C$, in which $A \bowtie B$ has been changed into its IVM-equivalent subtree.

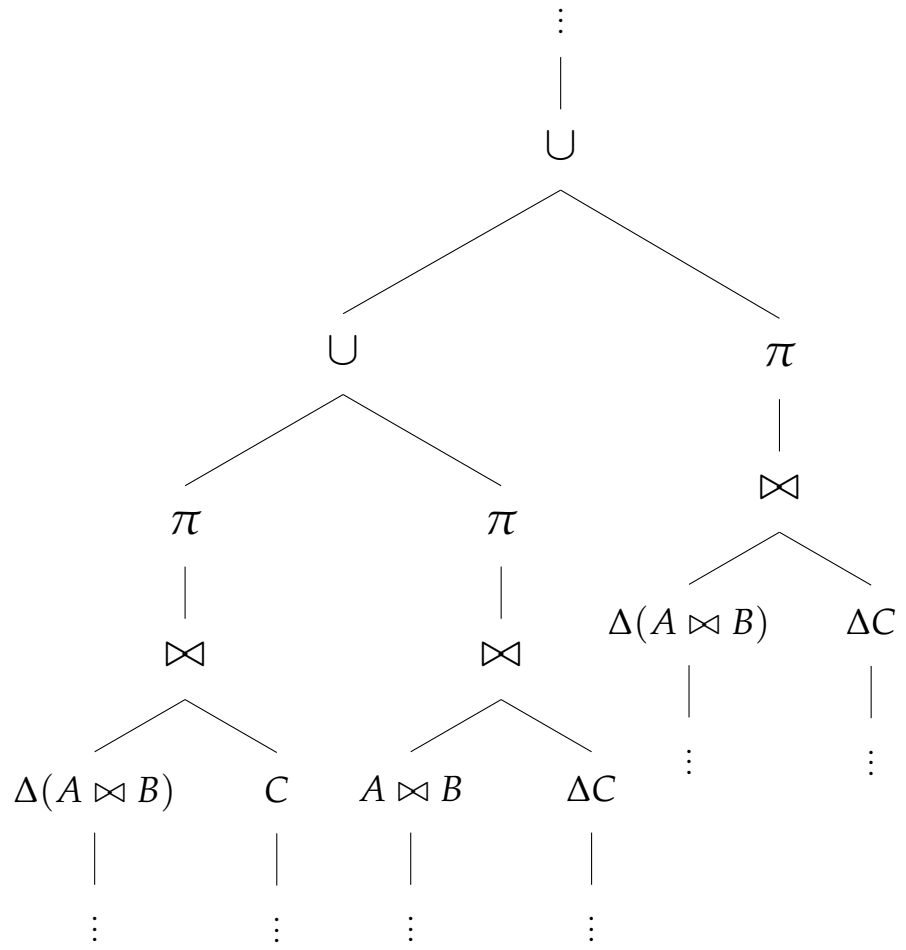


Figure 5.6.: Fully modified query tree for $(A \bowtie B) \bowtie C$, with truncated subtrees for $A \bowtie B$ and $\Delta(A \bowtie B)$ due to size constraints.

5.3. Effect on join conditions

This section aims to clarify whether join conditions need to be modified if delta tables are involved, and if so, what needs to happen in those instances.

For joins which have a delta table on only one side, the join conditions can stay intact. Compared to the non-IVM query, the output now includes a single additional multiplicity column. This has no consequences for the join itself, but may require an additional projection above the join operator.

The situation changes when both sides of a join consist of delta tables. In this case, both sides have a multiplicity column, meaning that the output of the join would contain two distinct multiplicity columns. Consequently, it may become ambiguous which multiplicity column prevails for the remainder of the query tree. Using the schema depicted in Subsection B.1.2, Figure 5.7 depicts some sample data that can be used to explain the behaviour of joins between two delta tables.

uid	user_name	multiplicity
1	Laika	1
2	Scooby	0
3	Toto	0
7	Lassie	0

Table 5.1.: The `delta_gods` table.

from_uid	to_uid	amount	multiplicity
4	1	10	0
3	1	12	1
1	2	4	0
7	3	3	1

Table 5.2.: The `delta_payments` table.

Figure 5.7.: Example data for ΔL (Δ`gods`) and ΔR (Δ`payments`), including the multiplicity columns. Here, a multiplicity value of 0 and 1 correspond to `false` and `true` respectively.

Suppose a query is executed in which the `uid` from the `gods` table is joined on `to_uid` from the `payments` table using the condition `gods.uid = payments.to_uid`. Let the `gods` table correspond to L, and the `payments` table to R. The join of interest for this explanation is then $\Delta L \bowtie \Delta R$ (which would actually get their data from `delta_gods` and `delta_payments` respectively). If the join conditions of $L \bowtie R$ were to be taken over without any modifications, the result set of this join would include all records in Table 5.3.

This result set now has two multiplicity columns, whereas the eventual result set of the logical plan mandates only one multiplicity column. Therefore, it is necessary to theorise what should be the multiplicity value for each of those records, and which of the records should appear in the result set.

The first record concerns an addition to ΔL matching a removal inside ΔR. In terms

uid	user_name	multiplicity	from_uid	to_uid	amount	multiplicity
1	Laika	**1**	4	1	10	**0**
1	Laika	**1**	3	1	12	**1**
2	Scooby	**0**	1	2	4	**0**
3	Toto	**0**	7	3	3	**1**

Table 5.3.: The result of joining the tables in Figure 5.7 on `gods.uid = payments.to_uid`, with the values of the multiplicity columns emboldened.

of the MV, this means that the records have not interacted previously, and will not interact after any propagation. Consequently, this match does not represent a record that existed in the MV prior to maintaining the MV, nor does it represent a record after maintaining the MV. Since these two records therefore effectively do not interact with each other, the resulting record should therefore not appear in the IVM result set.

The second record represents an addition to both ΔL and ΔR, which trivially leads to an addition to the MV. This record should therefore be included in the result set, with its multiplicity set to 1 (`true`).

The third record represents a deletion in both ΔL and ΔR, which – symmetrically to the previous record – trivially leads to a deletion from the MV. This record should therefore be included in the result set, but with the multiplicity set to 0 (`false`).

Finally, the fourth record concerns a removal inside ΔL matching an addition to ΔR. Like the first record, the records do not interact as this match does not represent a record in the MV either before or after this maintenance query. Thus, this record should be omitted from the result set.

uid	user_name	multiplicity	from_uid	to_uid	amount	multiplicity
1	Laika	**1**	3	1	12	**1**
2	Scooby	**0**	1	2	4	**0**

Table 5.4.: The result of $\Delta L \bowtie \Delta R$ after filtering redundant records based on multiplicity.

Based on aforementioned details, Table 5.4 depicts the records that $\Delta L \bowtie \Delta R$ should yield. What remains now is to determine how to handle the existence of two multiplicity columns. What can be observed is that the two multiplicity columns now bear identical values, meaning that there no longer is a dissonance between the two. This is logical: records across join sides can only interact with each other if they are either both additions or both removals. From this, two conclusions can be drawn.

First, $\Delta L \bowtie \Delta R$ should be extended with an additional join condition. Namely, the condition $\Delta L_{\texttt{multiplicity}} = \Delta R_{\texttt{multiplicity}}$ ensures that non-interacting join pairs are omitted.

Second, since the multiplicity columns are identical, it does not matter which one is forwarded to the join's ancestors. Either multiplicity column can be picked, as long as the resulting projection is compatible with the other joins in the IVM query.

5.4. Implementation

This section describes the practical implementation of joins in OpenIVM. The practical implementation follows the theoretical approach from the preceding sections. Inside DuckDB, the modification of the original logical plan is done by inserting new nodes into the logical plan, and modifying existing nodes where necessary.

As explained previously in Section 3.3, the logical plan is modified bottom-up. For queries without joins (or any other operators with two children), there is only one lineage to make modifications to (where a lineage is defined as any possible path from a leaf node to the root node). However, when joins are involved, multiple lineages can exist, which can complicate this recursive plan modification significantly.

Concretely, the biggest difference between implementing joins and the previous operators that OpenIVM already had implemented, is that part of the tree should retain its original state. More specifically, if the original query contains a join such as $L \bowtie R$, then the original subtree of L needs to exist in the tree once (for the join $L \bowtie \Delta R$), as well as the original subtree of R (for the join $\Delta L \bowtie R$), as reasoned in Section 5.1. For the practical implementation, this means that the traversal algorithm should make a copy of the nodes L and R before recursing further into the tree, to then later be able to use these copies in the additional joins.

Once the DFS has made all modifications to both the LHS and RHS of a join, the next step is to modify the join itself. The objective is to modify the original join – as depicted in Figure 5.1 – with three different joins whose output is combined by unions (as depicted in Figure 5.3). Since the LHS and RHS now represent ΔL and ΔR respectively, their subtrees are both duplicated to be used in two different joins.

Upon the creation of these three joins, their result sets have to be combined using two union operators. Because the column count and positioning should be identical for each column – as discussed in Section 5.1 – each join will have a projection as its parent. The implementation of this projection is specified in Subsection 5.4.2.

Once the logical plan is structured as depicted in Figure 5.3, the ancestors of topmost union – now at the position of the original join – have to be modified. Since a union (a `LogicalSetOperation` inside DuckDB) defines its own index, it also defines its own CBs.

As a consequence, the entire lineage from the union's parent up until (and including) the nearest ancestor that defines its own table index needs to have their CBs revised.

This operation is further complicated by the fact that the union is in place of the original join. In DuckDB, joins do **not** define their own table index, meaning that CBs in the original join can be derived from arbitrarily many table indices. For a correct modified plan, all previously used CBs need to be adjusted, thus making it necessary to keep track of all CB changes while the modifications are performed. To this and, a mapping is made which maps each superseded CB (as it appeared in the original join) to to its successor CB (in the topmost union). The remainder of the logical plan is modified as usual.

5.4.1. Table index conflicts

As mentioned in Subsection 4.2.3, DuckDB checks whether table indices are only used in one lineage. For each binary operator, it is permitted for descendants on only one side of the operator to utilise a specific table index. This constraint implies that – for each table index – there must be a unique path from the root to the deepest descendant that uses that index (in other words, where it is first introduced).

For the implementation of joins in OpenIVM, this means that any modification of the logical plan should maintain this property. The use of the same table index by descendants on both sides of a binary operator would normally not occur, but becomes an issue when duplicating arbitrary subtrees for joins.

When a subtree – in this case: the LHS or RHS of join – is copied, this yields three separate subtrees in a logical plan that are derived from each join side of the original $L \bowtie R$. Although L and R are supplanted twice (by ΔL and ΔR respectively), the table indices are not replaced in this procedure by default.

This problem was resolved by implementing a rigorous table index renumbering scheme. Fortunately, it was not needed to completely revise the logic of creating two copies of the join tree of $L \bowtie R$. Instead, the subtrees of four of the six resulting join sides – specifically, each join side corresponding to ΔL or ΔR – would have all table indices renumbered, such that table indices are once more distinct across the logical plan. The join sides corresponding to the original query – the RHS in $\Delta L \bowtie R$ and the LHS in $L \bowtie \Delta R$ – are left unmodified. The steps for renumbering the table indices are roughly as follows:

1. Traverse the subtree bottom-up, starting from any leaf nodes.
2. If an operator has its own table index, generate a new index whilst maintaining a mapping from new index to old index. The CBs defined by this operator should automatically adjust to this new table index.

3. Recursively modify the table index of operators in this subtree, until the root of the subtree is reached.

4. Using the mapping, correct all CBs in any expressions of the subtree.

5. Use this same mapping for the join conditions in the subtree's parent join operator.

This is a tedious yet necessary operation to make the plan pass the checks that DuckDB has put in place to ensure the integrity of query execution plans. To this end, table indices are modified in every defining operator, their expressions, and CBs.

Only after all table indices are renumbered, the new joins are created and inserted into the modified plan. All that remains from this point is to create a projection above the joins with the new CBs taken into account.

5.4.2. Handling multiplicity columns

The introduction of joins in OpenIVM also affects the handling of the multiplicity column. Since there no longer is a single lineage in the query, it became necessary to modify the OpenIVM logic for adding the multiplicity CB to each operator in the modified plan. The multiplicity CB is now explicitly returned by the recursion.

Previously, it was assumed that the multiplicity column was always the last column inside a (modified) operator. Thus, its respective CB was passed through implicitly. However, this logic is incompatible with a join output potentially having two different multiplicity CBs – and proved to be potentially unstable otherwise – meaning that the CB must be provided explicitly as a consequence. This change makes the multiplicity CB accessible for all operators that are specifically added to replace the original join.

For a proper handling of multiplicity CBs within the modified plan, the following steps are undertaken during the recursive join modification.

First, the multiplicity CBs of ΔL and ΔR are obtained using the ancestors of the join.[1] Using these CBs, the new projections (defined on top of the new joins) can precisely determine which of its input CBs correspond to a multiplicity column. Each projection then projects this CB as the last column of a result set.[2] Due to the newly created projections, both new unions can be assured that their last CB is the multiplicity column. Therefore, upon creating the topmost union, it can be safely assumed that this union's final CB corresponds to the multiplicity column. It is therefore this CB which will be forwarded to its parent operator as the multiplicity column.

[1] More precisely: the CBs are obtained in the mapping described in Subsection 5.4.1, using the ancestor's CBs as the key to query this mapping. The output of this mapping is then the replacement CB (or CBs) corresponding to one of the three newly created joins.

[2] For $\Delta L \bowtie \Delta R$, the projection also arbitrarily discards one of the multiplicity CBs.

6. Logical Plan to SQL

In this chapter, the functionality to convert a logical plan in DuckDB to a SQL query is discussed.

The purpose of compiling a logical query plan to SQL is to make execution of OpenIVM's modified plans possible on any RDBMS supporting SQL. Although this functionality is made specifically for OpenIVM, it could in theory be used for any application that needs a SQL representation of a logical plan.

First, Section 6.1 explains why converting a logical plan to a SQL query is a non-trivial problem. Then, Section 6.2 denotes the current functionality that DuckDB offers for converting operators – and by extension, logical plans – to SQL snippets. Consequently, the state of the logical plan's SQL representation before the implementation of joins in OpenIVM – as well as its shortcomings – is discussed in Section 6.3. Finally, Section 6.4 introduces Logical Plan to SQL (LPTsql), which is the new implementation for converting a plan into an executable SQL query.

6.1. Problem description

SQL queries can be arbitrarily complex, and so can their logical plan be. The main problem when creating a SQL compiler – to convert logical plan into a representative SQL string – is that it should also be capable of handling such complex plans.

Each logical plan for an MV is in itself based upon a SQL query. A possible approach may be to simply rewrite this query by adding delta tables were appropriate. However, the delicacies of the SQL language would make this an especially scrupulous task.

Consider an MV query containing a join. Not only would the query have be rewritten to use three joins; rewriting the original SQL query would also involve accounting for projections, join conditions, and many other corner cases that exist within SQL's syntax. If a direct SQL-to-SQL rewriter were to account for all such cases, it would need to perform tasks similar to ones that the optimiser already has implemented. Thus, the original SQL is of little to no use for a plan that is modified for IVM purposes.

This notion is best illustrated with an example. Suppose an MV is made for one of the generic queries in Subsection B.2.4, such as the query for an inner join. For convenience, this query is also depicted in Listing 6.1. As described in Section 5.1,

the original join ($L \bowtie R$) would have to be replaced by three different joins: $\Delta L \bowtie R$, $L \bowtie \Delta R$, and $\Delta L \bowtie \Delta R$.

```
SELECT * FROM table_l INNER JOIN table_r
ON table_l.x = table_r.y;
```

Listing 6.1: A query with an inner join and an explicit join condtion in its simplest form.

When changing the query to account for those three new joins, one would have to create three subqueries – one for each join – and adapt them for the delta tables. The result would be similar to Listing 6.2. Note that merely converting the base query into a subquery per join already requires special handling for $\Delta L \bowtie \Delta R$ and the additional join condition on the multiplicity (in accordance with Section 5.3). In this instance, it is possible to append it to the preceding join condition. However, for any query with more complicated base queries, finding the correct place to insert this additional join condition may require significant effort.

```
-- Subquery 1.
SELECT * FROM delta_l AS table_l INNER JOIN table_r
ON table_l.x = table_r.y
-- Subquery 2.
SELECT * FROM table_l INNER JOIN delta_r AS table_r
ON table_l.x = table_r.y
-- Subquery 3.
SELECT * FROM delta_l AS table_l INNER JOIN delta_r AS table_r
ON table_l.x = table_r.y AND table_l.multiplicity = table_r.multiplicity
```

Listing 6.2: A partial adaptation of the inner join query for IVM.

Not only the join condition is complicated to handle adequately. Due to the introduction of delta tables, each subquery now has a different set of columns. These columns need to be aligned with each other before the result set of each of those subqueries can be combined using a `UNION`. However, since the base query's columns are defined using '`SELECT *`', the SQL completely lacks any information on the columns – therefore requiring information external to the SQL itself to retrieve which columns should be projected.

Considering the above, it would be unreasonable to use the original SQL query as a basis for writing the IVM query. Not only would information external to the SQL query be necessary; the amount of corner cases across the query as a whole is significant enough that it may be best to discard the original query completely, and

instead exclusively utilise the modified logical plan to compose the IVM query.

It is necessary, therefore, to devise other approaches to convert the logical plan into a representative SQL query. The possibilities and available functionality to this end are explored in the forthcoming sections.

6.2. Current functionality in DuckDB

Currently, DuckDB offers no functionality to convert a logical query plan (a tree consisting of logical operators) into a representative SQL query. However, each individual expression (part of an operator) has limited support for expression that operator as a string.

On the development side, the behaviour (and output) of this functionality depends on whether the debugging mode is enabled, and thus the usefulness of these helper functions is often limited. A concrete example: if column bindings lack aliases, the debug mode would return its column bindings, whereas the non-debug mode would retrieve a (non-alias) column name. Consequently, if one wishes to have consistent behaviour regardless of whether development mode is enabled, this functionality (or significant parts of it) would have to be written from scratch.

6.3. Previous IVM query composition in OpenIVM

Prior the implementation of joins, OpenIVM already had the functionality to convert a logical plan to a SQL query (named `LogicalPlanToString`). However, this functionality was not able to support arbitrarily complex queries, among which IVM queries containing joins. The implementation was mostly tailored to rather simple queries, and only unary operators (along with a cross join) were supported. Concretely, the issues with `LogicalPlanToString` mostly concentrate around the fact that it was made for simple query plans only. For example, queries were assumed to have only few conditions.

To support joins in OpenIVM, the query plan uses both joins and unions, making their support in a SQL compiler a necessity. Initially, the intention was to extend the functionality of `LogicalPlanToString` with unions (and joins), however this ended up being impractical for multiple reasons. The principal reason is that the result of a union does not have a name to refer to (except if put in a subquery), but using a subquery for unions would not work well with said implementation. Furthermore, since both a join and a union are a binary operator, it was structurally not feasible to reasonably extend the current code with those operators.

6.4. Revised Logical Plan to SQL (LPTsql) implementation

As the preceding sections may have made clear, there is no satisfactory functionality either in DuckDB or in the OpenIVM extension to compile SQL queries for logical plans containing joins.

For this reason, it is necessary to discard all current functionality for compiling SQL queries, and instead introduce a new logic to compose a SQL query from the logical plan. This new logic is devised in this section.

First, the rationale behind LPTsql's approach is explained. The remaining part of this section then covers issues that have arisen during the implementation of this functionality, as well as how those issues were overcome.

6.4.1. Using CTEs to the rescue

Recall from Section 2.6 that every operator in RA has an equivalent representation in SQL, whether direct or indirect. From this fact it can be hypothesised that it might be possible to represent each operator in isolation as SQL.

The code for LPTsql was remade from scratch with a different approach: rather than 'linearly' shaping a SQL string, each operator in the query plan gets its own CTE. This functionality, named LPTsql, traverses the query tree and recursively creates CTE's using a bottom-up approach. This idea is explained using a relatively simple query. Given the original SQL query (in Listing 6.3), DuckDB generates the query tree depicted in Figure 6.1.

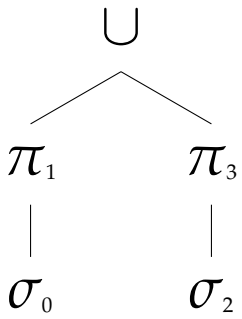


Figure 6.1.: Logical plan of the query in Listing 6.3, as generated by DuckDB. The digits next to the operators depict LPTsql's enumeration scheme.

```
1 SELECT * FROM sales UNION SELECT * FROM sales;
```

Listing 6.3: Example SQL query for LPTsql.

The first implementation of LPTsql would convert this logical plan to the 'operator-to-CTE' query in Listing 6.4. One may notice that in this particular SQL query, the projection CTEs are redundant and could be omitted. Ostensibly, there is no benefit in introducing unnecessary CTEs into a query, but – in line with Section 2.1 – as long as this method of generating a query always yields a syntactically and semantically correct query, an insignificant amount of redundant CTEs should not negatively affect performance.

```
with scan_0 AS
   (SELECT order_id, product_name, amount, date_ordered FROM sales),
projection_1 AS
   (SELECT order_id, product_name, amount, date_ordered FROM scan_0),
scan_2 AS
   (SELECT order_id, product_name, amount, date_ordered FROM sales),
projection_3 AS
   (SELECT order_id, product_name, amount, date_ordered FROM scan_2)
SELECT * FROM projection_1 UNION SELECT * FROM projection_3;
```

Listing 6.4: Output of LPTsql for the logical plan in Figure 6.1.

As mentioned previously, the query as depicted in Listing 6.4 is what the first version of LPTsql would output. However, there is an issue with this approach: if a join has identical columns on both its left and right side, then the creation of a CTE for each individual operator could cause naming conflicts between those two columns. A more detailed description for this problem, as well a solution, is devised in Subsection 6.4.3.

6.4.2. Intermediate Representation (IR)

LPTsql uses an Intermediate Representation (IR) to convert a logical plan into a SQL query. This IR stores each query operator as a *node*. The IR stores all nodes – except for the root of the logical plan – inside a vector. The insertion order for this vector is congruent to a DFS with bottom-up insertions. Each node in this vector is interpreted as a CTE – in insertion order – such that this vector represents the list of CTEs to be declared in the query.

In the current implementation, the IR is converted directly to SQL. While this is functional, it does not make the system robust across SQL dialects. However, LPTsql was devised in such a manner that it would able to eventually transform the IR to an AST. To this end, each node in LPTsql is structured in such a way that its properties can be easily used inside an AST. This AST, in turn, would become capable of being transformed into various SQL dialects.

The functionality to convert the IR directly to SQL works as follows. Each read-only node can be converted into both a normal (non-CTE) SQL substring, but also a stand-alone CTE. The CTE functionality effectively wraps the non-CTE query around CTE-specific syntax. If the IR has at least one CTE, the SQL query is initiated with the keyword 'WITH', after which the CTE representation of each non-root node is appended.

After appending all CTEs to the in-progress SQL query string, the topmost node (the root node) needs to be converted into the main part of the query. If a read-only node is the topmost operator of a tree, this is equivalent to using the non-CTE SQL substring, but potentially with the introduction of aliases. However, the final node can also be a write-operator (such as an `Insert`), which in turn only has one representation.

A simplified explanation of the SQL corresponding to a selection of nodes can be found in the Appendix under Appendix C.

6.4.3. A fix for column name conflicts

To overcome the column name conflict issue, the following system was devised: each CTE would explicitly name its columns according to the table index that they are defined in. This means that CTEs may still use the same column names from their ancestors, but if and only if their are derived from the same index-generating operator. It is expressly **not** the intention to use LPTsql's enumeration scheme for this, as this would not solve the issue for join operators.

Indeed, the children of a join would use column names with their table index embedded, which guarantees that they are unique, and therefore, cannot conflict with column names from the other join side. For example, a scan with table index 7 of a table with the columns `user_id, user_name` would become `t7_user_id, t7_user_name`. Subsequently, most CTEs would explicitly define the columns that they use.

Code-wise, each CB (e.g. `[7, 0], [7,1]` in the preceding example) is mapped to a bespoke data structure (struct) that contains the necessary properties for this functionality to work. The struct for the LPTsql implementation contains the table index, original column name, and an optionally defined alias. Whenever requested, this struct can then create a function with a guaranteed unique column name; namely, a column name that combines the table index with the original column name (or alias, if defined).

If a certain column binding is used in a non-CB-generating operator, it can access the map to see what it should be named. If the CB-generating operator (non-GET) has to define column names, it can use the ones of its descendants to create a new struct (and take over a name, if needed).

Specifically for joins, the specific naming of the columns (using the struct) ensures that no two columns will ever have an identical name. Without this, it is possible that

a column in one table has an identical name to a column in the other table. When matching keys – for example in a join – this indeed can occur quite often.

When these generated column aliases are used in a query *and* the topmost query returns a table, the topmost operator should convert this alias back to either their default name or the alias as defined in the original query. Without this, a user may be exposed to column names in the output of the query that are neither the default column names or any explicitly defined aliases by the user.

With the above changes in mind, Listing 6.5 depicts how Figure 6.1 would be converted to SQL.

```
WITH scan_0
(t1_order_id, t1_product_name, t1_amount, t1_date_ordered) AS
  (SELECT order_id, product_name, amount, date_ordered
  FROM ivma.main.sales),
projection_1
(t2_order_id, t2_product_name, t2_amount, t2_date_ordered) AS
  (SELECT t1_order_id, t1_product_name, t1_amount, t1_date_ordered
  FROM scan_0),
scan_2
(t8_order_id, t8_product_name, t8_amount, t8_date_ordered) AS
  (SELECT order_id, product_name, amount, date_ordered
  FROM ivma.main.sales),
projection_3
(t9_order_id, t9_product_name, t9_amount, t9_date_ordered) AS
  (SELECT t8_order_id, t8_product_name, t8_amount, t8_date_ordered
  FROM scan_2),
union_4
(t0_order_id, t0_product_name, t0_amount, t0_date_ordered) AS
  (SELECT * FROM projection_1 UNION SELECT * FROM projection_3)
SELECT
  t0_order_id AS order_id,
  t0_product_name AS product_name,
  t0_amount AS amount,
  t0_date_ordered AS date_ordered
  FROM union_4;
```

Listing 6.5: Output of LPTsql for the logical plan in Figure 6.1 after assigning explicit column aliases in each CTE.

7. Benchmarks

In this chapter, the implementation for joins in OpenIVM is evaluated. Section 7.1 describes the methodology behind the benchmarks, and Section 7.3 depicts the results.

7.1. Methodology

The easiest method to evaluate joins in OpenIVM is by comparing the performance of a query to its non-IVM equivalent. In Subsection B.1.3, the schema for the benchmark can be found, with the MV query depicted in Subsection B.2.3.
For the benchmark, the duration of the following operations are considered to count towards the execution time of the IVM query:

1. Creating the materialised view (and associated delta tables).
2. Creating an ART index (if applicable).
3. Inserting new data in the delta tables (of the base tables).
4. Compiling the IVM queries.
5. Inserting the result of the query defining the MV computed on the changes into the MV's delta table.
6. Merging changes from the MV's delta table into the MV itself.
7. Clearing the MV's delta table.

The first two operations are only executed in the first run of the benchmark. For each run, the execution time of all mentioned operations combined is compared to the execution time of the *reference query*, which consists of the 'main query' that defines the MV without the part that actually declares an MV (as depicted in Subsection B.2.3). The performance of this query measured upon completion of the IVM operations.

The rationale behind performing the reference query after the IVM operations is that the IVM can then take care of updating the data – whilst measuring its duration – without the reference query interfering with modifications to any tables.

7.1.1. Benchmark parameters

The bespoke benchmarking suite made for this evaluation consists of 5 parameters. Let L and R be the tables on the LHS and RHS of the query's join respectively. Then, the parameters to the benchmark are specified as follows, in order:

1. $|L|$: The initial record count in L.
2. $|R|$: The initial record count in R.
3. $|\Delta L|$: The amount of records added to L (through ΔL) in each run.
4. $|\Delta R|$: The amount of records added to R (through ΔR) in each run.
5. n: The entropy of the join condition, expressed as the amount of distinct values that each column in the a join condition can have.

All parameters are defined using absolute numbers. Using the parameters, sides of the join can be configured independently, such that different types of workload can be analysed. The entropy parameter further allows testing the change in performance based on the expected amount of matches per record, which in turn can affect the size of the MV relative to the base tables.

7.1.2. Analysed workloads

For the benchmarks, two workload groups were tested. In the first workload group – referred to as the *symmetrical table size* – L and R start with an identical amount of records. Furthermore each update adds an identical amount of records to each table.

The second workload – referred to as *facts and dimensions* – more closely resembles an OLAP workload with a lookup table. In this workload, L and R are initiated with a record count having a ratio of $1 : 100$ relative to each other, and each step only inserts new values in L.

7.2. Optimiser issue

An unexpected result from running the benchmarks was that, apparently, DuckDB is unable to optimise the query deleting entries from the MV – based on entries in the MV's delta table – in case there are few to no deletions. Consider the query in Listing 7.1 for deleting entries from the MV, as generated by OpenIVM:

```
1 delete from simple_join where exists
2 (
```

```
3   select 1 from delta_simple_join
4   where simple_join.increment_id = delta_simple_join.increment_id
5   and simple_join.dummy_name = delta_simple_join.dummy_name
6   and simple_join.tid_left = delta_simple_join.tid_left
7   and simple_join.geo_id = delta_simple_join.geo_id
8   and simple_join.dummy_location = delta_simple_join.dummy_location
9   and simple_join.tid_right = delta_simple_join.tid_right
10  and _duckdb_ivm_multiplicity = false
11 );
```

Listing 7.1: The query generated by OpenIVM to delete queries from the benchmark MV.

In an insertion-only workload, the column `_duckdb_ivm_multiplicity` would be true for every single column in `delta_simple_join`. Consequently, no record from `delta_simple_join` appears the parenthesised clause, and therefore, no matches can exist in the parenthesised clause. This in turn implies that `WHERE EXISTS` yields an empty result set, and thus no deletions have to be made.

All that the optimiser needs to recognise is that the query can never yield any deletions if `delta_simple_join` is empty (for `_duckdb_ivm_multiplicity = false`), and thus does not have to compute anything further. DuckDB is unable to recognise this optimisation, and instead executes the query using a 'right duplicate elimination join', which thus takes up the vast majority of execution time for MV updates. Even when the delta table contains a marginal amount of records to delete, such a rigorous join is not an efficient means of deleting records from the MV.

Rewriting this query using a CTE, as depicted in Listing B.6 (in the Appendix), did not improve the query's performance, indicating that this is not a mere formulation issue. Only when constructing the query using the relatively uncommon `USING` keyword – such as in the query in Listing B.7 (also in the Appendix) – does the optimiser recognise that only very few records (if any) are to be matched, and optimises thusly.

7.3. Results

All values in the benchmark depict the execution time in milliseconds, as executed on a bare metal device running Linux with 64 GB of Random Access Memory (RAM), 8 CPU cores, and 16 threads. Since a workaround was found for the optimisation issue in Section 7.2, the performance of the benchmarks in this section are not affected by this.

Symmetrical table size

In Table 7.1, the initial MV perform a join with 20 million records on each join side. From there, each iteration of the benchmark adds 2 million changes, corresponding to an increase of 10% per step in the benchmark. Since $n = 20 \cdot 10^6$, each record on the LHS is expected to join with only one record on the RHS. Since the reference implementation is relatively fast under this workload, the IVM implementation is unable to outperform it. When the join yields a larger result set – namely, when n is lowered while the other parameters otherwise stay intact – the two implementations have a more similar runtime. This is shown in Table 7.2. When n is lowered further, much larger result set. The benchmark depicted Table 7.3 uses $n = 600\,000$ – which is less than n in Table 7.3 – and this simple difference makes the IVM join outperform the reference join. Thus, for a workload in which many join partners are expected, the threshold at which a full refresh is more beneficial lays higher than for queries with a lower rate of join partners.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Using IVM	1 343	567	768	850	863	975	957	970	997	982
Reference	219	224	221	226	235	276	252	251	273	276

Table 7.1.: Benchmark for $|L| = 2 \cdot 10^7$, $|R| = 2 \cdot 10^7$, $|\Delta L| = 2 \cdot 10^6$, $|\Delta R| = 2 \cdot 10^6$, and $n = 2 \cdot 10^7$.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Using IVM	5 796	1 224	1 087	1 267	1 181	1 308	1 287	1 150	1 171	1 174
Reference	624	632	646	658	668	696	709	723	743	771

Table 7.2.: Benchmark for $|L| = 2 \cdot 10^7$, $|R| = 2 \cdot 10^7$, $|\Delta L| = 2 \cdot 10^6$, $|\Delta R| = 2 \cdot 10^6$, and $n = 2 \cdot 10^6$.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Using IVM	16 744	1 593	1 611	1 574	1 565	1 689	1 649	1 699	1 767	1 787
Reference	1 881	1 934	1 979	2 016	2 086	2 107	2 135	2 209	2 253	2 280

Table 7.3.: Benchmark for $|L| = 2 \cdot 10^7$, $|R| = 2 \cdot 10^7$, $|\Delta L| = 2 \cdot 10^6$, $|\Delta R| = 2 \cdot 10^6$, and $n = 6 \cdot 10^5$.

Facts and dimensions workload

A similar approach was taken when analysing the facts and dimensions workload. For these workloads, $|L|$ is the larger table which hence is the only table with insertions after the initial MV creation. In the following benchmarks, the ratios between the parameters are fixed, and hence the only difference lies in the amount of data processed.

In Table 7.4, IVM does not outperform the reference implementation. When all values are doubled however – as is seen in Table 7.5 – there is a turnover point after the 5th run in which the IVM join starts outperforming the reference join. This specific benchmark may be interesting to determine where the precise cutoff point lies between an IVM refresh and a full refresh. As may be expected, increasing the values further (as shown in Table 7.6) results in IVM completely outperforming the reference implementation after the first run.

To further illustrate the rate of changes for which using IVM is worthwhile Figure 7.1 depicts a series of benchmarks for which only $|\Delta L|$ is modified. When performance of the reference join and IVM join are compared at the tenth iteration of the benchmark, this graph shows that IVM has a better performance until the amount of changes for these parameters are between 8 and 20 million records. This corresponds to a cutoff point somewhere between 2%-5% for these specific parameters.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Using IVM	5 791	1 390	1 034	1 028	964	1 011	986	989	993	1 077
Reference	716	762	744	766	802	777	812	802	831	869

Table 7.4.: Benchmark for $|L| = 2 \cdot 10^8$, $|R| = 2 \cdot 10^6$, $|\Delta L| = 4 \cdot 10^6$, $|\Delta R| = 0$, and $n = 2 \cdot 10^6$.

Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Using IVM	10 081	1 835	1 930	1 986	1 829	**1 631**	**1 568**	**1 633**	**1 604**	**1 548**
Reference	**1 536**	**1 602**	**1 641**	**1 640**	**1 723**	1 774	1 725	1 781	1 806	1 839

Table 7.5.: Benchmark for $|L| = 4 \cdot 10^8$, $|R| = 4 \cdot 10^6$, $|\Delta L| = 8 \cdot 10^6$, $|\Delta R| = 0$, and $n = 4 \cdot 10^6$. The faster variant of each run is emboldened.

Run	1	2	3	4	5	6	7	8	9	10	11	12
Using IVM	14 487	2 452	2 421	2 452	2 610	2 535	2 637	2 516	2 203	2 333	2 285	2 287
Reference	2 535	2 505	2 544	2 587	2 618	2 670	2 723	2 782	2 809	2 854	2 907	2 952

Table 7.6.: Benchmark for $|L| = 6 \cdot 10^8$, $|R| = 6 \cdot 10^6$, $|\Delta L| = 1.2 \cdot 10^7$, $|\Delta R| = 0$, and $n = 6 \cdot 10^6$.

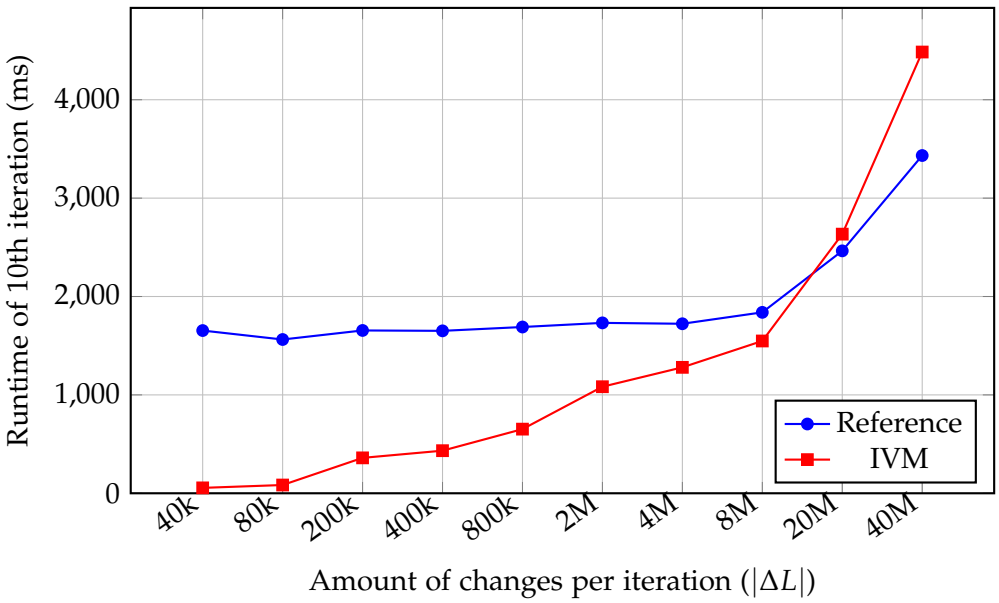


Figure 7.1.: Benchmark for $|L| = 4 \cdot 10^8$, $|R| = 4 \cdot 10^6$, and $n = 4 \cdot 10^6$, for varying $|\Delta L|$.

Part III.

Outlook

8. Other join types

This chapter aims to review how other join types could be implemented in OpenIVM.

The implementation of inner joins in OpenIVM has already been discussed extensively in this thesis, and it is thus clear that IVM can support inner joins. The joins of interest are therefore the other join types that were discussed in Section 2.5.

Some of the join types – such as semi-joins and anti-joins – already have their IVM instructions devised in DBSP [14]. In DBSP, an anti-join $L \triangleright R$ is performed using $L - (L \ltimes R)$, in which the same L is input to both the semi-join and the set difference operator.

It is therefore more interesting to discuss the join types that have not been previously covered. These are mostly the system-specific join types used in HyPer [9] and DuckDB [15]. Before delving into some of these special join types however, the more 'conventional' outer join types (left, right, full) are analysed first.

8.1. Outer joins

Relative to inner joins, the logic for outer joins in IVM is slightly more complicated, due to the maintenance of both join matches and unmatched records (for either or both join sides).

If a record in the MV currently represents a match, but this match is removed during maintenance, it takes non-trivial effort (without materialisation) to figure out whether there are still other join partners. This is not a problem for workloads in which deletions are rare. However, if deletions and inserts are similarly common, materialisation may be necessary to ensure efficient IVM for these join types. There would be one materialisation for matching join partners, and one for unmatched records (two in the case of a full outer join, as each join side has its own schema).

In the modified logical plan, the query tree would not be as symmetrical as it would be for inner joins. The reason for this is that there are different join types needed to semantically convey the intention of the outer join types. Outer joins introduce the concept of non-matches (and therefore `NULL`s for any records that cannot find a join partner), and the MV may only contain a singular representation of each unmatched record. This is in contrast to three IVM joins sources in which a match can occur; a record should appear as unmatched in the MV if and only if **none** of the combinations

between L, ΔL, R, and ΔR yield a match. This property could significantly complicate the modified plan, especially if the intention is to not materialise any intermediate results. More specifically, a record in L (or a new record in ΔL) is only really unmatched if **no** match exists in any of $L \bowtie \Delta R$, $\Delta L \bowtie R$, $\Delta L \bowtie \Delta R$, as well as the current state of the MV.

Disregarding the state of the current MV, this means that for any join in which a new unmatched record could appear, there are two other joins in which they can nevertheless be paired (which means the unmatched record is to be discarded). From the perspective of the current state of the MV, there exist three joins in which a join partner can appear for any currently unmatched record. Creating records for the unmatched and subsequently discarding them is tedious and would be messy to implement. Rather, execution using the following steps is proposed:

1. Materialise an intermediate table for $L \bowtie R$ (called M in the remaining steps).
2. Materialise an intermediate table for $L \triangleright R$ (called N in the remaining steps).
3. When running an update, calculate the *positive* changes as done for inner joins (depicted as M^+).
4. Shrink N using $N \leftarrow N - \pi_{N.*}(M^+)$.
5. For the negative changes (removals) to matches: distinguish changes that are exclusively caused by removals in ΔR from changes in which a removal on ΔL's side (also) plays a role. This means separating $L \bowtie \Delta R$ from the other two delta-joins. The reason for this is that every pending removal in ΔL can never result in an addition to the unmatched intermediate table (N). In RA, the removals from the MV result set that are a consequence of removals from R could be represented as $(L \bowtie \Delta R)^-$.
6. Grow N by adding the following: $N \leftarrow \pi_{N.*}(L \bowtie \Delta R)^- - \pi_{N.*}((L \bowtie R) - (L \bowtie \Delta R))$. Effectively, this means: add the records that were just removed to the unmatched set (N), excluding the records which still have other partners in the MV (and therefore in M).
7. Finally, (re)construct the MV using $M \cup N$, with the necessary `NULL`s assigned on the columns derived from N where applicable.

A right outer join is functionally identical, but with the sides inverted. A full outer join would also require similar logic, but with three intermediate tables ($L \bowtie R$, $L \triangleright R$, and $R \triangleright L$) and the rest adapted accordingly.

8.2. Mark joins, and single joins

Mark join

The output of a mark join is equivalent to all records of the LHS of a join, with an additional column depicting existence of a matching entity on the RHS. If changes to each side of the join are considered in isolation, it may be possible to reason about how changes to those tables affect the view of interest. Let L represent the LHS, and R the RHS. Then, changes to L and R have the following consequences:

- $\Delta(L)^+$: Positive changes to L. An addition of a record to the LHS automatically means an addition of a record to the result set. The existence quantifier is determined by the current state of the RHS, in conjunction with the delta state of the RHS.
- $\Delta(L)^-$: A deletion of a record from the LHS. Any deletion of a record in L results in a deletion of the respective record from the MV's result set. No further computations need to be made; this deletion is therefore trivial.
- $\Delta(R)^+$: An addition of a record to the RHS can either change the existence boolean to true (a record on the LHS now has a match whereas it did not previously) or do nothing (there is no match on the LHS, or one or more other records on the RHS already matched the respective record(s) on the LHS). In any case, it does not change the quantity of records in the result set; only some booleans may be altered.
- $\Delta(R)^-$: A removal of a record on the RHS may mean that one or more records on their LHS lose their (last) match from the RHS. This in turn may change the existence booleans corresponding to the respective result tuple to `FALSE`. In other cases, where the record removal on the RHS is a 'redundant' record – from the perspective of existence checks – the output set does not change.
- Special cases when both are modified: Care needs to be taken when an addition to ΔL coincides with a removal of a match in ΔR. If the change in ΔR corresponds to the only matching record for the addition on the LHS, the existence quantifier must be set to `FALSE`. Other pairs of changes should not require special treatment.

In short: when `NULL`s are not a factor, mark joins can theoretically be supported by IVM. However, to implement it, it may perhaps become necessary to maintain a counter (in combination with an aggregate) for the amount of matching records, rather than a simple boolean depicting 'match exists' or 'no match exists'. Without such a counter,

the arithmetic concerning the existence quantifier may become too complex to viably be supported in IVM.

An alternative would be to implement a mark join in a similar style to outer joins, by materialising an intermediate result set with all matches, and another intermediate result set of all non-matching columns. Then, the maintenance step of IVM would first update these intermediate result sets, after which it can merge the changes to update the remainder of the view.

Single join

Although a single join could be implemented as a variant to an outer join, this would dismiss a great potential to optimise for them in IVM. Since each record on the RHS of a single join can have either 0 or 1 match on the RHS, any record appearing in the delta table for the MV has a specific semantic:

- If a specific addition and deletion correspond to the same join condition, this must mean that a scalar in the MV should be replaced with another scalar.
- If an addition corresponds to a join condition – but without a deletion matching the same join condition – then this means an unmatched record in the MV must become matched.
- Finally, if a deletion corresponds to a join condition (without an addition matching the same join condition), then this means an unmatched record in the MV must become matched.

This logic is significantly simpler than the logic that generic outer joins have to implement. Thus, single join operators could therefore benefit from having a dedicated implementation in IVM.

9. Conclusion

After having extensively discussed joins in OpenIVM, this conclusion reflects on the contents of this thesis. For this, the research questions in Chapter 1 are revisited, and potential future work is discussed.

9.1. Research questions revisited

How can join functionality be added to OpenIVM?
Implementing joins in OpenIVM is not as straightforward as it may seem on the surface. This thesis mostly focussed on inner joins, which already appeared to be rather involving to implement. Although it should be possible to implement all other join types in OpenIVM as well, it remains to be seen for which join types this actually yields significant performance benefits in IVM workloads.

Under what workloads is IVM on queries with an inner join beneficial?
Several workloads were tested on samples of various sizes, in which both the ratio of records between two the two join sides and the expected amount of matches per record were alternated. For non-trivial base table sizes, the implementation of inner joins in OpenIVM generally outperforms inner joins without IVM if the updates made to the data are equivalent to less than 5% of the base table data. The higher the average amount of join matches is for each record on a join side, the higher the cut-off point is for where IVM no longer outperforms a non-IVM implementation.

Is it viable to convert arbitrary query plans generated by DuckDB into SQL, and if so, how?
Previously, the SQL compiler of OpenIVM lacked support for joins and was tailor-made for simpler queries, which made it unclear whether arbitrarily complex queries could be supported. As mentioned previously in Section 6.2, as of present, there is no official functionality in DuckDB to convert a logical query plan into a SQL query.

The logic for converting a query plan to SQL was completely revised for this thesis. As a consequence, it was possible to create a much more rigorous foundation for this functionality. This code (LPTsql) became surprisingly robust, making it rather straightforward to support operators such as casts without too much hassle.

As a result, the logic for converting the query plan to SQL could potentially support all operators and hence become a stand-alone functionality of DuckDB – rather than the current state of effectively being pegged to the IVM implementation.

9.2. Future work

A great quality of life improvement for OpenIVM would be to complete the AST implementation of LPTsql. Currently, the LPTsql code directly converts its IR into a SQL query, which is simpler to implement at the cost of flexibility. This should make it possible to execute and test OpenIVM on database systems using different SQL dialects than DuckDB. In fact, it might be possible to create a dedicated extension for LPTsql's functionality, without explicitly being tied to OpenIVM, such that it can be used for other (research) projects inside the DuckDB ecosystem.

In terms of joins, one straightforward future work would be to implement other join types in OpenIVM, as well as to analyse their performance. Furthermore, it would be worthwhile to devise adaptive materialisation strategies for joins in OpenIVM.

On a related note, it would be interesting to benchmark the OpenIVM join implementation on other systems, perhaps assisted by first finishing the aforementioned AST functionality of LPTsql.

Specifically for DuckDB, the optimiser should be reviewed to figure out why certain deletion queries are not executed efficiently.

When it comes to DuckDB-specific optimisations for joins in OpenIVM, it could be worth looking to determine if there is perhaps a way to reuse scans when creating the IVM joins. The behaviour of CTEs inside DuckDB may give an insight in how this could potentially be achieved. Currently, the modified query plan for an IVM query with joins contains two separate scans for each individual delta-table. One scan is for the join between the delta-subtree and the base subtree (i.e. $\Delta L \bowtie R$ or $L \bowtie \Delta R$) and the other for the mutual delta join ($\Delta L \bowtie \Delta R$). It should be possible to implement an optimisation – either in the DuckDB optimiser or in LPTsql – which merges two separate table scans together if they access the same base table with an identical selection (and ordering) of columns. These conditions should hold for both ΔL and ΔR, as their columns (and any potential projections) are chiefly determined by the original IVM query.

A. Overview of selected relational algebra operators

The following tables show operators in RA that are frequently used across this document.

Operation	Symbol
Selection	σ
Projection	π
Aggregation	Γ
Natural Join	$\bowtie$
Union	$\bigcup$
Intersection	$\bigcap$
Set Difference	$-$

Table A.1.: Common RA operators.

Join Type	Notation
Inner join	$L \bowtie R$
Equi-join	$L \bowtie_{L.x=R.y} R$
Left outer join	$L \mathbin{⟕} R$
Right outer join	$L \mathbin{⟖} R$
Full outer join	$L \mathbin{⟗} R$
Semi-join	$L \ltimes R$
Anti-join	$L \triangleright R$
Cross product	$L \times R$

Table A.2.: Common join operators in their RA notation. L and R are relations representing the LHS and RHS of the join.

B. Queries and schemata

B.1. Schemata for OpenIVM testing and benchmarking

B.1.1. Original OpenIVM schema

The following table was used to demonstrate IVM for queries without joins.

```
CREATE TABLE sales (
  order_id INT, product_name VARCHAR(1), amount INT, date_ordered DATE
);
```

B.1.2. Schema for OpenIVM with joins

To implement and test IVM with joins, the schema consisting of the following two tables was used:

```
-- Base tables.
CREATE TABLE gods (uid INT, user_name TEXT);
CREATE TABLE payments (from_uid INT, to_uid INT, amount INT);
```

Listing B.1: Schema used to test and implement joins in OpenIVM.

B.1.3. Schema for join benchmarks in OpenIVM

```
-- Base tables.
CREATE TABLE left_side (
  increment_id INTEGER, dummy_name VARCHAR, tid_left INTEGER
);
CREATE TABLE right_side (
  geo_id INTEGER, dummy_location VARCHAR, tid_right INTEGER
);
```

Listing B.2: Schema used to benchmark the OpenIVM implementation for joins.

B.2. OpenIVM queries

B.2.1. OpenIVM demonstration queries

The base functionality in OpenIVM (for queries using only one base table) is demonstrated using the following materialised views.

```
CREATE MATERIALIZED VIEW product_sales AS
  SELECT
    product_name,
    SUM(amount) AS total_amount,
    COUNT(*) AS total_orders
  FROM sales
  WHERE product_name = 'a' OR product_name = 'b' GROUP BY product_name;

CREATE MATERIALIZED VIEW product_sales AS
  SELECT *
  FROM sales
  WHERE product_name = 'a';

CREATE MATERIALIZED VIEW product_sales AS
  SELECT SUM(amount) AS total_amount
  FROM sales;
```

Listing B.3: Queries used to demonstate the base OpenIVM implementation.

B.2.2. OpenIVM join queries

The implementation of joins in OpenIVM was tested using the following materialised views.

```
-- All payments, annotated with the sender's name.
CREATE MATERIALIZED VIEW named_payments AS
    SELECT g.user_name, p.from_uid, p.to_uid, p.amount
    FROM gods AS g, payments AS p
    WHERE p.to_uid = g.uid;

-- All payments from uid 3, including the amount.
CREATE MATERIALIZED VIEW recipients AS
  SELECT g.user_name, p.amount
```

```
  FROM gods as g, payments as p
  WHERE p.from_uid = 3 AND g.uid = p.to_uid;

-- Sum of incoming payments made by a god.
CREATE MATERIALIZED VIEW income AS
  SELECT g.user_name, sum(p.amount)
  FROM gods as g, payments as p
  WHERE p.to_uid = g.uid
  GROUP BY g.user_name;
```

Listing B.4: Queries used to test the OpenIVM joins implementation.

B.2.3. Benchmark query

The following query was used to benchmark joins in OpenIVM. Since the benchmark compares the performance of an IVM query to its non-IVM counterpart, a comment delineates which part of the query is exclusive to the MV.

```
-- View query.
-- MV part.
CREATE MATERIALIZED VIEW simple_join AS
-- Main query.
SELECT * FROM left_side INNER JOIN right_side ON tid_left = tid_right;
```

Listing B.5: Query used to benchmark the OpenIVM implementation for joins.

The queries below are rewrites of Listing 7.1 as an attempt to resolve optimiser issues.

```
WITH cte AS (
  SELECT * from delta_simple_join WHERE _duckdb_ivm_multiplicity = false
)
delete from simple_join WHERE EXISTS (
  SELECT 1 from cte
  WHERE simple_join.increment_id = cte.increment_id
  AND simple_join.dummy_name = cte.dummy_name
  AND simple_join.tid_left = cte.tid_left
  AND simple_join.geo_id = cte.geo_id
  AND simple_join.dummy_location = cte.dummy_location
  AND simple_join.tid_right = cte.tid_right);
```

Listing B.6: An alternative version of OpenIVM's delete query for the benchmark's MV, using a CTE.

```
WITH cte AS (
  SELECT *
  FROM delta_simple_join
  WHERE _duckdb_ivm_multiplicity = false
)
DELETE FROM simple_join
USING cte
WHERE simple_join.increment_id = cte.increment_id
  AND simple_join.dummy_name = cte.dummy_name
  AND simple_join.tid_left = cte.tid_left
  AND simple_join.geo_id = cte.geo_id
  AND simple_join.dummy_location = cte.dummy_location
  AND simple_join.tid_right = cte.tid_right;
```

Listing B.7: An alternative version of OpenIVM's delete query for the benchmark's MV, using the `USING` keyword.

B.2.4. Expressing different join types in SQL

For some of the join types in Subsection 2.5.1, a direct SQL representation exists. Given two tables L and S (represented in SQL as `table_l` and `table_r`), and some condition (here: `table_l.x = table_r.y`), the following SQL queries mention their join type explicitly:

Inner join `SELECT * FROM table_l INNER JOIN table_r ON table_l.x = table_r.y;`
This particular case is also an equi-join.

Left join `SELECT * FROM table_l LEFT JOIN table_r ON table_l.x = table_r.y;`

Right join `SELECT * FROM table_l RIGHT JOIN table_r ON table_l.x = table_r.y;`

Outer join `SELECT * FROM table_l FULL OUTER JOIN table_r ON table_l.x = table_r.y;`

Cross join `SELECT * FROM table_l CROSS JOIN table_r;`

Although it is not possible to express all join types explicitly in SQL, some join types are the result of a more complex query. For example, a semi-join and anti-join may be used for the queries in Listing B.8.

```
1 -- Example semi-join.
2 SELECT table_l.* FROM table_l JOIN table_s ON table_l.x = table_s.y;
3 -- Example anti-join.
4 SELECT table_l.* FROM table_l
5 EXCEPT
6 SELECT table_l.* FROM table_l JOIN table_s ON table_l.x = table_s.y;
```

Listing B.8: Example queries for a semi-join and anti-join.

In this particular example, one may notice that the anti-join query is identical to the semi-join query after the `EXCEPT` keyword, making the nomenclature of an anti semi-join clear. Although the latter join types (semi-join and anti-join) do not have an explicit representation in SQL, distinguishing them from the other join types may be useful for optimisation purposes. This is in line with the additional join types introduced by HyPer, which do not have an explicit SQL representation either [9].

C. Representation of operators in LPTsql

This chapter includes a simplified specification of each of the nodes in LPTsql, as well as its IR.

C.1. Individual nodes

The following sections contains example representation of a selection of operators in LPTsql. Each operator is represented as a *node*, and the syntax used below is a mix of regular expressions and SQL.

Representation of `CteNode`

Every node but the topmost node is represented as a `CteNode`. Each read-only node (corresponding to an operator) has a CTE representation, but write-only operators do not.

```
cte_name
(col[, col_i ...])
AS (
derived_node_query
)
```

Representation of `GetNode`

```
SELECT
* | col[, col_i ...]
FROM
catalog.schema.table_name
[WHERE table_filter[, table_filter ...]]
)
```

Representation of `FilterNode`

```
SELECT * FROM
child_cte
[WHERE table_filter[, table_filter ...]]
```

Representation of `GetNode`

```
SELECT
* | col[, col_i ...]
FROM
child_cte
```

Representation of `AggregateNode`

```
SELECT
[groupby_col[, groupby_col_i ...]]
aggr_col[, aggr_col_i ...]
FROM
child_cte
GROUP BY
[groupby_col[, groupby_col_i ...]]
```

Representation of `JoinNode`

```
SELECT * FROM
left_cte
INNER|LEFT|RIGHT|OUTER
JOIN
right_cte
ON
join_condition [AND|OR join_condition ...]
```

Representation of `UnionNode`

```
SELECT * FROM
left_cte
UNION [ALL]
SELECT * FROM
right_cte
```

Representation of `InsertNode`

```
INSERT [OR REPLACE|IGNORE]
INTO table_name
SELECT * FROM
child_cte_name
```

C.2. Intermediate representation (IR)

The IR of LPTsql is structured as an object containing any amount of `CteNodes`, and precisely one `FinalNode`. The non-AST implementation of LPTsql converts the IR into SQL query roughly as follows.

```
[WITH cte_node [, cte_node ...]]
final_query;
```

Abbreviations

ART Adaptive Radix Tree

AST Abstract Syntax Tree

CB Column Binding

CPU Central Processing Unit

CTE Common Table Expression

DFS Depth-First Search

HTAP Hybrid Transactional/Analytical Processing

IR Intermediate Representation

IVM Incremental View Maintenance

ISO International Organization for Standardization

LHS left-hand side

LPTsql Logical Plan to SQL

MV Materialised View

OLAP Online Analytical Processing

OLTP Online Transaction Processing

RA Relational Algebra

RAM Random Access Memory

RDBMS Relational Database Management System

RHS right-hand side

SIMD Single Instruction, Multiple Data

SQL Structured Query Language

List of Figures

List of Tables

List of Listings

Bibliography

[1] I. Battiston, K. Kathuria, and P. Boncz, *OpenIVM: A SQL-to-SQL Compiler for Incremental Computations*, arXiv:2404.16486 [cs], Apr. 2024. DOI: 10.1145/3626246.3654743.

[2] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970, ISSN: 0001-0782. DOI: 10.1145/362384.362685.

[3] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., Mar. 2017, ISBN: 978-1-4493-7332-0.

[4] ISO/IEC, *ISO/IEC 9075-1:2023: Information technology — Database languages SQL*, en, 2023.

[5] P. Boncz, M. Zukowski, and N. Nes, "MonetDB/X100: Hyper-Pipelining Query Execution," in *Proceedings of the 2005 CIDR Conference*, Jan. 2005.

[6] T. Neumann and M. 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.

[7] SQLite, *SQL Features That SQLite Does Not Implement*, Documentation.

[8] D. Sotolongo, D. Mills, T. Akidau, A. Santhiar, A.-P. Tóth, I. Battiston, A. Sharma, B. Huang, B. Zhang, D. Pauliukevich, E. Sartorello, I. Belianski, I. Kalev, L. Benson, L. Papke, L. Geng, M. Uhlar, N. Shah, N. Semmler, O. Zhou, S. Nowak, S. Lionheart, T. Merker, V. Lifliand, W. Grus, Y. Huang, and Y. Zhu, *Streaming Democratized: Ease Across the Latency Spectrum with Delayed View Semantics and Snowflake Dynamic Tables*, arXiv:2504.10438 [cs], Apr. 2025. DOI: 10.1145/3722212.3724455.

[9] T. Neumann, V. Leis, and A. Kemper, "The complete story of joins (in hyper)," *Datenbanksysteme fur Business, Technologie und Web, BTW 2017 - 17. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme, DBIS 2017, Proceedings*, Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft fur Informatik (GI), B. Mitschang, D. Nicklas, F. Leymann, H. Schoning, M. Herschel,

J. Teubner, and T. Harder, Eds., pp. 31–50, 2017, Publisher: Gesellschaft fur Informatik (GI).

[10] T. Neumann, "Query simplification: Graceful degradation for join-order optimization," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09, New York, NY, USA: Association for Computing Machinery, Jun. 2009, pp. 403–414, ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559889.

[11] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann, "Adopting worst-case optimal joins in relational database systems," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 1891–1904, Jul. 2020, ISSN: 2150-8097. DOI: 10.14778/3407790.3407797.

[12] D. D. Chamberlin, "Early History of SQL," *IEEE Annals of the History of Computing*, vol. 34, no. 4, pp. 78–82, Oct. 2012, ISSN: 1934-1547. DOI: 10.1109/MAHC.2012.61.

[13] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, *DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views*, arXiv:1207.0137 [cs], Jun. 2012. DOI: 10.48550/arXiv.1207.0137.

[14] M. Budiu, F. McSherry, L. Ryzhyk, and V. Tannen, *DBSP: Automatic Incremental View Maintenance for Rich Query Languages*, arXiv:2203.16684 [cs], Mar. 2022. DOI: 10.48550/arXiv.2203.16684.

[15] M. Raasveldt and H. Mühleisen, "DuckDB: An Embeddable Analytical Database," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 1981–1984, ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3320212.