

Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Computer Science

Bringing Flummi to WITH MUTUALLY RECURSIVE on Materialize DB

Ludwig KOLESCH

25.07.2024

Examiner

Prof. Dr. Torsten GRUST

Supervisor

Tim FISCHER

Ludwig KOLESCH:

Bringing Flummi to WITH MUTUALLY RECURSIVE on Materialize DB

Bachelorthesis Computer Science

Eberhard Karls Universität

From 01.04.2024 to 25.07.2024

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Ludwig KOLESCH

Abstract

Flummi is a compiler that translates programs written in the simple, imperative language PL\Flummi into the Structured Query Language (SQL). It converts each block of a program's Control Flow Graph (CFG) into a Common Table Expression (CTE) and implements loops in the control flow using a recursive CTE. In this thesis, we adjust Flummi to compile PL\Flummi programs into *mutually* recursive CTEs for evaluation by Materialize DB, a streaming SQL database system with its own specialized version of recursive CTEs. We then examine the performance impact of this change in compilation method on PL\Flummi programs, discovering that Materialize DB performs better on compiled queries that leverage its mutual recursion capabilities than on those using only standard SQL recursion. A key feature of Materialize DB is its support for incremental updates, allowing for recomputing only parts of the query results when underlying data changes, rather than reprocessing the entire data. We explore the performance of PL\Flummi programs with incremental updates and our experiments show that the queries perform well in these dynamic scenarios. Despite the generally slower performance of Materialize DB compared to other Database Management Systems (DBMSs) like PostgreSQL, we conclude that PL\Flummi programs can benefit from mutual recursion, and that in combination with incrementalization, Materialize DB is a viable compilation target for Flummi in real-world applications.

Contents

Abstract	v
Acronyms	ix
1. Introduction	1
2. Concepts	3
2.1. Flummi	3
2.2. Materialize DB	5
2.2.1. WITH MUTUALLY RECURSIVE	5
2.2.2. Incremental Updates in Materialized Views	8
3. Implementation	11
3.1. Compiling PL\Flummi to SQL	11
3.2. The Loopless Rule	12
3.3. Generating Mutually Recursive Queries	13
3.4. Adhering to Syntax of Materialize DB	13
4. Performance Tests	15
4.1. Test Setup	15
4.2. Test Results	16
4.2.1. Performance of Mutual Recursion	16
4.2.2. Performance of Incremental Updates	19
4.2.3. Issues with Materialize DB	20
4.3. Interpretation	20
5. Conclusion	23
5.1. Summary	23
5.2. Future Work	23
A. Appendix: Measure Charts	25
Bibliography	29

Acronyms

CFG Control Flow Graph
CTE Common Table Expression
DBMS Database Management System
SQL Structured Query Language
UDF User-Defined Function

Introduction

An established principle in database systems research advises executing all computations near the data to enhance efficiency. Ideally, complex computations on data should be performed within the DBMS itself, to avoid the overhead of copying all the data into another program's memory, performing the calculations there, and possibly copying the updated data back. However, formulating complex queries in SQL can be challenging, because if a looping control flow is required, you need to use recursion. There are two ways to perform recursion in a DBMS: With a recursive User-Defined Function (UDF), which tends to be inefficient for larger amounts of data, or using a recursive CTE, introduced in the SQL:1999 standard [1]. While recursive CTEs are more performant, they are often harder to read and write than UDFs, as suggested by Duta in a user study from 2020 [2].

To address this challenge, the Database Chair at the University of Tübingen has developed Flummi, a compiler that translates programs written in the simple imperative language PL\Flummi into pure recursive SQL CTEs. This allows you to write complex queries with comprehensible control flow constructs such as variable assignment, loops, and branching, while still running the computations directly inside a DBMS without the performance drawbacks of a UDF.

The Flummi compiler works by analyzing a PL\Flummi program and creating a CFG consisting of connected blocks that can assign variables, compute and emit values. Each of these blocks is then converted into a SQL CTE, which can reference variables assigned in previous blocks via the **FROM** clause. To implement loops in the control flow, the entire query is placed inside a recursive CTE that is repeatedly evaluated until a termination condition is met, propagating computed variables back to the beginning of the loop.

Materialize DB [3], a streaming SQL database system, implements a special kind of recursive CTEs with fewer restrictions, called **WITH MUTUALLY RECURSIVE**: Unlike in most DBMSs, multiple CTEs may mutually reference each other and are computed in order repeatedly. For Flummi, this eliminates the need for the outer recursive CTE to propagate computed values back: Instead, the block-CTEs become mutually recursive and can reference values directly, even when they are computed further down in the CFG. In this thesis, we explore how to adjust Flummi to generate mutually recursive queries suitable for Materialize DB and measure the resulting performance benefits.

Another key feature of Materialize DB is its support for incremental updates: When a complex query is run on some data, and a small part of that data changes, the results of the query can be updated accordingly without the need to reevaluate the query over the entire dataset. This is achieved by storing the query results in a *materialized view*. We test the performance of incremental updates for compiled PL\Flummi programs in materialized views to measure their

efficiency compared to other DBMSs that need to reevaluate the query over the whole dataset each time.

The objective of this thesis is to augment the Flummi compiler to support mutually recursive queries for Materialize DB, compare the performance of these queries with those using standard recursion, and evaluate their performance in materialized views. Through this process, we aim to determine the suitability of Materialize DB for executing PL\Flummi queries and identify areas for improvement, both for the Flummi compiler and for Materialize DB.

Concepts

First, we will take a look at Flummi, Materialize DB, and how PL\Flummi queries can potentially benefit from the features of Materialize DB.

2.1. Flummi

Flummi is a compiler for the simple language PL\Flummi, developed by the Database Systems Research Group at the University of Tübingen. It provides straightforward control flow mechanisms and compiles to a SQL query which can be executed by a relational DBMS. The goal of the Flummi compilation method is to demonstrate how imperative control flows can be transformed into recursive queries that are efficiently executed by a DBMS. This approach can be particularly efficient to evaluate the same code for many different inputs, leveraging the DBMS's optimization for parallelization.

Since the Flummi compiler targets SQL, PL\Flummi is designed to be very simple, providing only control flow without any data-related operations. Instead, all computations of values are performed using SQL embedded within the PL\Flummi source code. These computed values can then be assigned to variables or output from the program. When the program is compiled, the embedded SQL calculations integrate with the compiled query. This way, PL\Flummi only needs to provide sequential statements, declaration and assignment of variables, branching, loops, output of values, and halting.

When a PL\Flummi program is compiled, the Flummi compiler parses the code and generates a minimal CFG composed of blocks of code. Each block is then converted into a CTE. If a CTE assigns a variable that is referenced by another CTE, the first CTE is placed above the second in the order of evaluation. During its computation, it outputs a row directed to the second CTE that contains the variable's assigned value. The second CTE can then reference this value using the SQL **FROM** clause.

However, for loops, this approach is insufficient. If CTE *A* refers to a variable of CTE *B*, but in the next loop iteration CTE *B* must read the updated variable from CTE *A*, placing one CTE above the other will not work: Neither of them can be computed independently. The solution is to enclose all these block CTEs in a large, *recursive* CTE **LOOP**, which repeatedly evaluates all block CTEs in order until the termination condition is met. This allows CTE *A* to reference variables from CTE *B* as before, and output the updated variables into the loop. These updated values are then available to CTE *B* in the next loop iteration, enabling the proper looping control flow.

To manage this, the Flummi compiler distinguishes between **GOTO**s and **JUMP**s in the generated CFG: A **GOTO** is a simple edge from one block to another, while a **JUMP** is a back edge used when

the control flow needs to be passed back up to a CTE earlier in the CFG. **JUMP**s therefore represent cycles in the control flow: Only if the program contains loops, one CTE might need to be executed repeatedly to read a variable modified by itself or another CTE further down the graph. In such cases, a **JUMP** row is generated by the latter CTE and made available to the former by the **LOOP** CTE. Flummi’s compilation of imperative control flows into SQL queries is described in detail in the paper “SQL Engines Excel at the Execution of Imperative Programs” [4].

The syntax of PL\Flummi is shown by the example program **Mul** in Listing 2.1:

```

1 CALL ($4$, $5$) IN
2 FUN (a: $int$, b: $int) -> $int$: {
3   c: $int; -- variable c is declared (as the SQL type `int`)
4   c <- $0$; -- variable c is set (to the SQL expression `0`)
5
6   LOOP multiplication_loop { -- code loops until BREAK or STOP
7     c <- ${0} + {1}$[a, c]; -- addition in SQL using variables a and c
8     EMIT ${0}$[c]; -- output current value of c
9     b <- ${0} - 1$[b];
10    IF $0 = {0}$[b]
11      THEN BREAK multiplication_loop
12      ELSE CONTINUE multiplication_loop
13  };
14  STOP
15 }

```

Listing 2.1: Example PL\Flummi program: **Mul**

This program defines a function that takes two integer inputs, *a* and *b*, and returns a set of integers. It multiplies the inputs by repeatedly adding *a* to the variable *c* and emits each intermediate value of *c* as a result. The variable *b* is decremented in each step, so when *b* reaches 0, the multiplication is complete and the loop exits, terminating the program. The function is called with the values 4 and 5, producing the resulting values 4, 8, 12, 16, and 20. Expressions enclosed in between paragraph symbols (\$) are SQL expressions for which the compiler inserts the specified variables for the placeholders ({#}) and integrates them into the resulting compiled query.

Figure 2.1 shows the generated blocks during compilation: The loop from the program is converted into a **JUMP** between two blocks. Each block will then be represented by a CTE, with the enclosing **LOOP** CTE required to pass the control flow from **inter1** back to **multiplication_loop_head** unless the termination condition $0 = b$ is met.

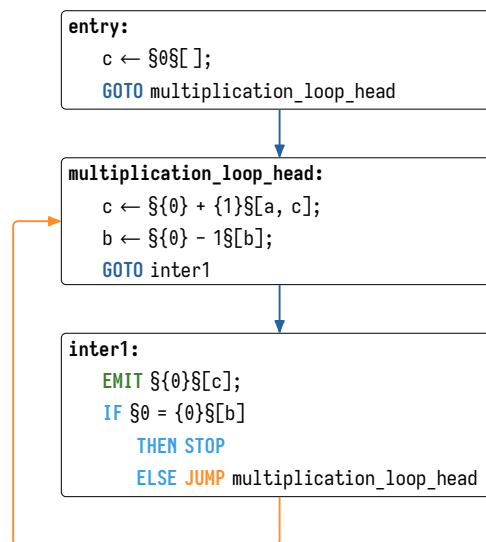


Figure 2.1.: Computed CFG blocks for **Mul**

2.2. Materialize DB

Materialize DB [3] is described as a “data warehouse purpose-built for operational workloads where an analytical data warehouse would be too slow, and a stream processor would be too complicated” [5]. It provides a SQL interface that is largely compatible with PostgreSQL, and has two features that might be of interest for PL\Flummi queries: Mutual recursion through its **WITH MUTUALLY RECURSIVE** syntax and incremental updates of computed results through *materialized views*.

2.2.1. WITH MUTUALLY RECURSIVE

Unlike other available DBMSs, Materialize DB allows mutual recursion in recursive CTEs. For example, consider the SQL query in Listing 2.2:

```
1 WITH RECURSIVE
2 even(e) AS (
3     SELECT 0
4     UNION ALL
5     SELECT o + 1
6     FROM odd
7     WHERE o < 6
8 ),
9 odd(o) AS (
10    SELECT e + 1
11    FROM even
12    WHERE e < 6
13 )
14 SELECT e FROM even;
```

Listing 2.2: Mutually recursive query

```
1 WITH MUTUALLY RECURSIVE
2 even(e integer) AS (
3     SELECT 0
4     UNION ALL
5     SELECT o + 1
6     FROM odd
7     WHERE o < 6
8 ),
9 odd(o integer) AS (
10    SELECT e + 1
11    FROM even
12    WHERE e < 6
13 )
14 SELECT e FROM even;
```

Listing 2.3: Same query in Materialize DB

The semantics of the query should be as follows: The **even** CTE includes 0 and all successors of **odd** numbers below 6, while the **odd** CTE includes all successors of **even** numbers below 6. When evaluated recursively, **even** should contain the numbers 0, 2, 4, and 6, and **odd** should contain the numbers 1, 3, and 5. However, PostgreSQL rejects the query from Listing 2.2, saying that mutual recursion between **WITH** elements is not implemented [6]. Similarly, other DBMSs either explicitly state in their documentation that mutual recursion is unsupported, such as MySQL [7], or simply do not support it without any mention in their documentation, such as SQLite3 [8] and Microsoft SQL Server [9]. Although the SQL:1999 standard defines mutual recursion, it neither specifies whether mutual recursive queries are legal nor clarifies their semantics [1].

Materialize DB does not implement **WITH RECURSIVE** queries as defined by the SQL standard. Instead, it has its own flavor of recursive queries, called **WITH MUTUALLY RECURSIVE**, which allows CTEs to mutually reference each other [10]. A revised version of the query from Listing 2.2 is shown in Listing 2.3. In addition to using the keyword **MUTUALLY**, Materialize DB also requires

explicit column types in recursive CTEs, so the columns `even.e` and `odd.o` need to be annotated as `integers`.

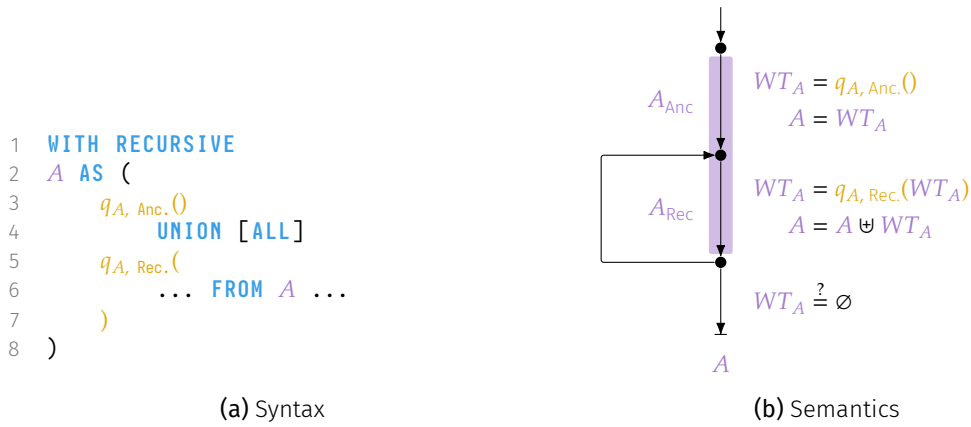


Figure 2.2.: Recursive CTEs in standard SQL

As shown in Figure 2.2, a recursive CTE A as defined by the SQL standard consists of two parts: the anchor part A_{Anc} and the recursive part A_{Rec} . The anchor part defines the start of the recursion by selecting initial rows as the base case, which are stored in the working table WT_A as well as in the final result A . The recursive query, connected to the anchor query with `UNION [ALL]`, can then select additional rows while referencing existing rows in WT_A . `UNION ALL` semantics are depicted in (b); `UNION` semantics can be achieved by defining $WT_A = q_{A, Rec.}(WT_A) \setminus A$ instead. The evaluation of A_{Rec} is repeated until WT_A is empty and no more rows are added to A , at which point the computation terminates and all produced rows form the resulting table A . Using this method, DBMSs typically allow only one recursive CTE to be computed at a time, thus disallowing mutual recursion.

Mutual recursion as defined in Materialize DB [11] is illustrated in Figure 2.3: This example defines three CTEs: A , B , and C . The syntactic requirement for separate anchor and recursive parts is removed, any valid `SELECT` expression is a valid recursive CTE query. All CTEs start as empty sets of rows and are evaluated repeatedly in order, up to an iteration where none of them change. The Δ operator represents the symmetric difference: $A_t \Delta A_{t+1}$ collects all rows that are in A_t but not in A_{t+1} or vice versa. Only when all CTE tables A , B , and C remain unchanged in an iteration, the computation of the three CTEs is finished.

The example in Listing 2.3 therefore works in Materialize DB. Table 2.1 illustrates the table contents after each iteration: Initially, both tables `even` and `odd` are empty. In the first iteration, `even` gets the value \emptyset , but no other values as `odd` is still empty. Then `odd` gets the value 1 by incrementing the value \emptyset now contained in `even`. As the tables have changed, the queries are evaluated again: `even` still contains \emptyset , but now also includes 2 by incrementing the value 1 now contained in `odd`. In turn, `odd` also includes 3 by incrementing this 2. These steps repeat until the fourth iteration when `odd` no longer changes because the `WHERE` predicate filters out the only new value, 6, from `even`. Since `even` changed in the fourth iteration, the evaluation of the recursive

CTEs still continues. In the fifth iteration, however, neither table changes, so the computation terminates.

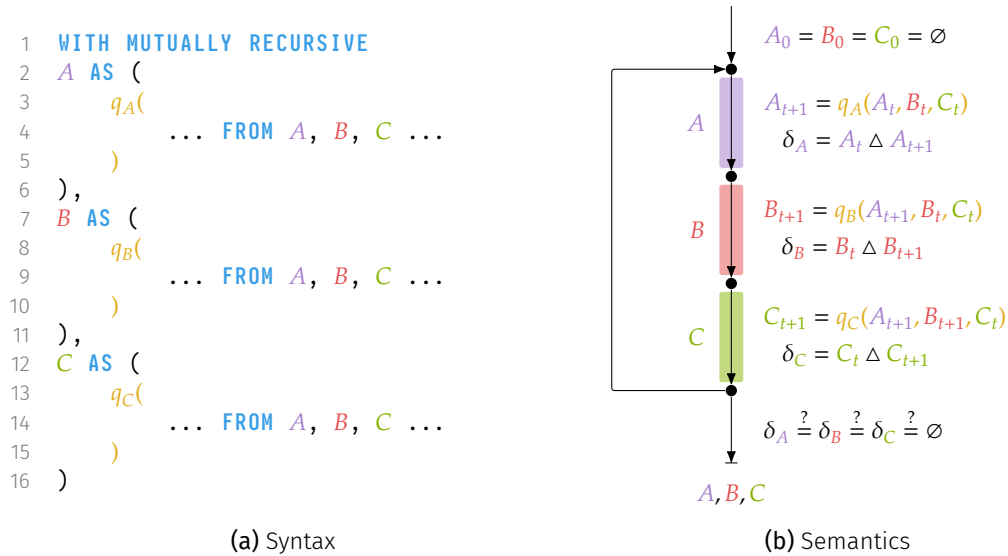


Figure 2.3.: Mutual recursive CTEs in Materialize DB

Iteration	even.e	odd.o
0	{}	{}
1	{0}	{1}
2	{0, 2}	{1, 3}
3	{0, 2, 4}	{1, 3, 5}
4	{0, 2, 4, 6}	{1, 3, 5}
5	{0, 2, 4, 6}	{1, 3, 5}

Table 2.1.: Evaluation of Listing 2.3 in Materialize DB

PL\Flummi programs could benefit from mutual recursion because it removes the need for **JUMP**s: Any CTE can generate **GOTO** rows for any other CTE, because even in the case of loops, the CTEs can refer to each other. This eliminates the necessity of the enclosing **LOOP** CTE that propagates variables back in loops.

Figure 2.4 (a) shows an adaptation of Figure 5 from “SQL Engines Excel at the Execution of Imperative Programs” [4]: The blocks \mathbb{b}_i and \mathbb{b}_j are mutually dependent within a loop, so they are evaluated repeatedly. Their corresponding CTEs, generated by the Flummi compiler, are CTE_i and CTE_j . Since CTE_j is placed below CTE_i , it can directly access the output of CTE_i in its **FROM** clause. However, CTE_i cannot directly access the output of CTE_j , so this output is passed on as a **JUMP** for the **LOOP** CTE to collect and provide to CTE_i in the next iteration.

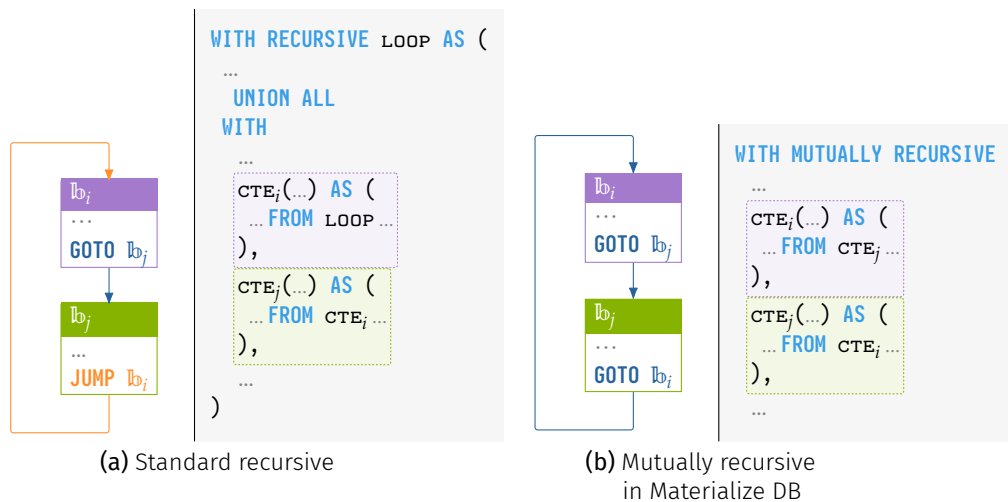


Figure 2.4.: CTE-based block chaining

Figure 2.4 (b) demonstrates how this can be done in Materialize DB. Since CTEs are allowed to mutually reference each other, both blocks b_i and b_j can call each other with `GOTO`s, resulting in mutual references in the `FROM` clauses of their respective CTEs. The semantics of `WITH MUTUALLY RECURSIVE` replace the `LOOP` CTE, allowing both block CTEs to be evaluated in order repeatedly and access each other's outputs. Data can therefore always be accessed from directly where it is generated, even in the presence of loops in the control flow, rather than being passed through the enclosing `LOOP` CTE first. Materialize DB's `WITH MUTUALLY RECURSIVE` is explicitly designed for such data flow, so we anticipate performance improvements as a result.

2.2.2. Incremental Updates in Materialized Views

Another special feature of Materialize DB is its materialized views. In a standard SQL view, a stored query transforms persistent data for other queries to read from. Each time the view is accessed, its stored query is evaluated to provide results that are up-to-date based on the underlying persistent data. This incurs the performance cost of having to evaluate the stored query each time the view is accessed. PostgreSQL also supports *materialized* views, which store both the query and its resulting data. While this eliminates the performance overhead of evaluating the stored query on each access, materialized views in PostgreSQL no longer provide data that is always up-to-date with the persistent table contents. Instead, the contents of the materialized view are only refreshed by explicitly using the `REFRESH MATERIALIZED VIEW` command, making it similar to a temporary table created from a view: Results are computed only once and persisted in tabular form, with the ability to repeat the same process later to update the contents.

Materialize DB introduces a different kind of materialized views that combine the advantages of standard and materialized views discussed above: The results of the view are persisted and

changes in the underlying table data are tracked. When the data changes, the materialized view is automatically updated. This is achieved through *differential dataflow* [12] which propagates only the differences between two table states through the view computation, updating only the parts of the view affected by the change. As a result, it is not necessary to recompute the entire view when only small changes occur in the data, because parts of the view unaffected by the changes are not recomputed. Since small changes in the data often affect only a small part of the view, this has great potential to improve performance, especially for large views over extensive datasets.

PL\Flummi programs that operate on data persisted in tables could also benefit from this feature, because if the only small portions of the data change, it is likely that only small parts of the program output will need updating. Evaluating the program as a materialized view could therefore lead to significant performance improvements when re-evaluating over slightly modified data.

Implementation

To understand my implementation of mutual recursion in the Flummi compiler, we will first look at the program-level SQL code generation, for programs with and without loops.

3.1. Compiling PL\Flummi to SQL

When compiling a PL\Flummi program to SQL, the Flummi compiler first transforms the code into n blocks b_1, \dots, b_n . Next, it maps these blocks to their respective SQL CTEs cte_1, \dots, cte_n . Each cte_i reads data from rows with a LABEL of ' b_i ' (for $i \in \{1, 2, \dots, n\}$) and can generate **JUMP** and **GOTO** rows for other CTEs to read from, and **EMIT** rows to form the program result.

```

1  WITH RECURSIVE LOOP(KIND, LABEL, <columns>, RESULT) AS (
2      (SELECT 'JUMP', 'b1', <program inputs>, NULL)
3          UNION ALL -- recursive union
4      (WITH CTE1(KIND, LABEL, <columns>, RESULT) AS ( ... ),
5          :
6          CTEn(KIND, LABEL, <columns>, RESULT) AS ( ... )
7
8      SELECT * FROM CTE1 WHERE KIND IN ('JUMP', 'EMIT')
9          UNION ALL
10         :
11         UNION ALL
12         SELECT * FROM CTEn WHERE KIND IN ('JUMP', 'EMIT')
13     )
14 )
15 SELECT RESULT FROM LOOP WHERE KIND='EMIT';

```

Listing 3.1: Program-level code generation with **JUMPs**

The further compilation procedure depends on the presence of **JUMPs** in the CFG: For programs with **JUMPs**, the standard Flummi compilation rule, shown in Listing 3.1, is applied: The program input is initially passed to the **LOOP** CTE in line 2 as a single **JUMP**. The CTE cte_1 corresponding to the entry block b_1 processes this input, producing **GOTO**, **JUMP**, and **EMIT** rows according to the program's CFG. All other CTEs are then evaluated based on the data currently available to them. Subsequently, the subquery in lines 8 through 12 selects all **JUMP** and **EMIT** rows generated by the block CTEs and collects them in the **LOOP** CTE. **GOTO** rows do not need processing by the **LOOP** since they are read directly by the targeted block CTE.

If new rows are added to **LOOP**, indicating new **JUMPs** or **EMITs** from the current iteration, the semantics of recursive CTEs ensure that the **LOOP** is reevaluated with the new rows. The new

JUMP rows are then available to their respective block CTEs in the next iteration. Once no more new rows are added to the **LOOP**, the computation is finished, and all **EMIT** rows generated during any iteration are selected by the query in line 15 as the result of the program.

3.2. The Loopless Rule

If there are no loops in the control flow and consequently no **JUMPs** in the CFG, the query can be simplified: The enclosing **LOOP** CTE, which was used to propagate **JUMP** rows back to their target block CTEs, becomes unnecessary for the control flow. The code generation is adjusted as shown in Listing 3.2: while a CTE called **LOOP** is still generated, its sole purpose is now to select the program input. It is no longer recursive and does not enclose the program blocks since there are no **JUMP** rows to propagate back to previous block CTEs. The inputs are still selected as a **JUMP** row for the entry block b_1 for consistency, but no other jumps are present in the generated code. All block CTEs are computed only once, based solely on the data provided to them by their predecessors through **GOTO** rows. Once they are all computed, the query in lines 11 through 15 directly collects all generated **EMIT** rows from the block CTEs to form the program result.

```
1 WITH
2   LOOP(KIND, LABEL, <columns>, RESULT)
3     AS (SELECT 'JUMP', 'b1', <program inputs>, NULL),
4
5   CTE1(KIND, LABEL, <columns>, RESULT)
6     AS ( ... ),
7   :
8   CTEn(KIND, LABEL, <columns>, RESULT)
9     AS ( ... )
10
11  SELECT RESULT FROM CTE1 WHERE KIND='EMIT'
12  UNION ALL
13  :
14  UNION ALL
15  SELECT RESULT FROM CTEn WHERE KIND='EMIT';
```

Listing 3.2: Loopless rule: program-level code generation without **JUMPs**

The detection of loopless queries and this resulting simplification of compilation were already part of the compiler when I began working on it. Any DBMS should benefit from this when executing loopless programs, as the query planner does not have to handle the constant re-evaluation of all block CTEs within the recursive **LOOP** CTE. Instead, knowing that all CTEs will be evaluated only once allows for more optimizations in their evaluation.

3.3. Generating Mutually Recursive Queries

The generation of mutually recursive queries can be achieved very similarly to this loopless rule and is shown in Listing 3.3: Without **JUMPs**, all CTEs can be top-level, making the enclosing loop unnecessary. The **LOOP** CTE again only selects the program inputs for the entry block b_1 . The main difference from actual loopless programs is that the block CTEs can now mutually reference each other, making the entire query loop according to Materialize DB's semantics.

```
1 WITH MUTUALLY RECURSIVE
2   LOOP(KIND text, LABEL text, <typed columns>, RESULT <result type>)
3     AS (SELECT 'JUMP', 'b1', <program inputs>, NULL),
4
5   CTE1(KIND text, LABEL text, <typed columns>, RESULT <result type>)
6     AS ( ... ),
7   :
8   CTEn(KIND text, LABEL text, <typed columns>, RESULT <result type>)
9     AS ( ... )
10
11 SELECT RESULT FROM CTE1 WHERE KIND='EMIT'
12    UNION ALL
13    :
14    UNION ALL
15 SELECT RESULT FROM CTEn WHERE KIND='EMIT';
```

Listing 3.3: Program-level code generation for Materialize DB

To implement this generation of mutually recursive queries, I added a new compiler flag, **USE_MUTUAL_RECURSION**, which replaces all **JUMPs** in the generated CFG with **GOTOs**. Almost no other changes were required: without **JUMPs** in the CFG, the existing loopless rule was triggered, promoting all block CTEs to top-level. They read from their **GOTO**-predecessors using **FROM** clauses, resulting in mutual references in the case of loops in the CFG.

Unlike before, the loopless compilation can now be called with a graph of **GOTOs** that is not acyclic. Since **JUMPs** in the graph represent back edges, replacing them with **GOTOs** will create a cycle of **GOTO** edges. This makes it impossible to place each block CTE below all of its **GOTO**-predecessors, as the cycles create mutual predecessors. For this reason, the order of the blocks in the graph is ignored when the **GOTO**-graph is cyclic, and all block CTEs are placed in an arbitrary order. Due to the **WITH MUTUALLY RECURSIVE** semantics, they can all access each other regardless of their position, so the query remains intact.

3.4. Adhering to Syntax of Materialize DB

To generate syntactically correct Materialize DB queries, I added a second compiler flag, **MATERIALIZEDB_FLAVOR**, which causes the compiler to use **WITH MUTUALLY RECURSIVE** instead of **WITH [RECURSIVE]** whenever loops are used, and to add the type of each column to the column lists for recursive CTEs. Since all parameter types, variable types and the return type need to

be specified in PL\Flummi, the compiler already has access to the required information. The columns `KIND` and `LABEL` are both always set to `text`.

By combining the `USE_MUTUAL_RECURSION` and `MATERIALIZEDB_FLAVOR` flags into a new DBMS-specific flag set called `materialize`, PL\Flummi code can now be compiled into Materialize DB-compatible SQL queries that utilize its mutual recursive capabilities, as shown in Listing 3.3. It is also possible to specify only the `MATERIALIZEDB_FLAVOR` flag, which will generate queries with `JUMPs` and an enclosing `LOOP` CTE as usual, but using `WITH MUTUALLY RECURSIVE` and typed CTE columns. These can also be run on Materialize DB, which does not support standard SQL `WITH RECURSIVE`, allowing us to compare the performance of mutually recursive queries against standard recursive queries with `JUMPs` in the same DBMS.

Performance Tests

With the compiler now generating syntactically and semantically correct SQL queries for Materialize DB, we can proceed to test their performance. To evaluate whether using mutual recursion offers advantages over the enclosing `LOOP` CTE, I compared both methods in Materialize DB and tested the same queries in PostgreSQL for reference. Additionally, I tested incremental updates of PL\Flummi programs in Materialize DB to determine if it can save time when the program input changes only slightly.

4.1. Test Setup

For the performance tests, I measured the execution time of several compiled PL\Flummi programs on Materialize DB. Most of these queries operate on data from the TPC-H benchmark suite for database systems and are actually subqueries of the real TPC-H benchmark queries [13]. To generate the data, I did not use the built-in TPC-H load generator of Materialize DB [14], since it does not allow to manually change the table contents later. Instead, I used DuckDB's TPC-H extension [15] and imported the generated data into the Materialize database.

TPC-H data can be generated with a specific scale factor, where a scale factor of 1 corresponds to approximately 1 GB of data. For my tests, I used a database with a scale factor of 1, but varied the number of input rows by preselecting only a certain number of rows and then passing those rows to the compiled query. Since Materialize DB does not support `PRIMARY KEY` or `UNIQUE` constraints, I created indices on all columns declared as potential keys in the TPC-H specification. These indices could enhance query performance by speeding up lookups on key columns, which are typically used for joining multiple tables.

I used the following PL\Flummi programs for testing on TPC-H data:

- **Late:** Calculates the number of orders per supplier for which they are responsible for delays.
- **Margin:** Determines the price margin that can be achieved for each order by substituting parts with cheaper equivalents.
- **Packing:** Finds a way to pack the line items of each order into a minimal number of fixed-size containers.
- **Ship:** Computes the most frequently used mode of transportation (ground, air, or mail) per customer. This query does not use loops and is therefore non-recursive, unlike the others.
- **Supply:** Calculates potential savings for each order by sourcing each line item from the cheapest available supplier.

I also tested with two additional queries that do not operate on TPC-H data:

- **Force**: Computes the force applied to a number of bodies using a Barnes-Hut tree.
- **VM**: Emulates a simple virtual machine with a small instruction set to compute the Padovan sequence [16].

The tests were conducted using a local installation of Materialize DB on a Linux machine with 64 GB of RAM and an Intel i7-7700 processor with 4 physical cores and 8 logical threads. The version of Materialize DB used was v0.96.0-dev, commit `e7ac79eaa1` [17], and CockroachDB ran in a Docker container at version 22.2.0. All tests used the default settings of Materialize DB without configuration changes. For comparison, I also executed the compiled queries on a PostgreSQL 16.1 instance in a Docker container on the same machine.

4.2. Test Results

The tests performed fall into two categories: evaluating the performance of mutual recursion compared to standard recursion, and assessing the efficiency of incremental updates.

4.2.1. Performance of Mutual Recursion

To evaluate the performance of mutual recursion, I compiled the PL\Flummi programs into SQL in two different ways: first, with both the `MATERIALIZEDB_FLAVOR` and the `USE_MUTUAL_RECURSION` flags to enable mutual recursion in Materialize DB; second, with only the `MATERIALIZEDB_FLAVOR` flag, using the standard Flummi compilation including `JUMPs`. This comparison of both compiled queries within Materialize DB allows us to measure the true impact of using mutual recursion, without any other changing factors. In this context, I will refer to the queries that use mutual recursion as “mutually recursive queries” and the queries with `JUMPs` as “standard recursive queries”, although both kinds of queries use the `WITH MUTUALLY RECURSIVE` syntax in Materialize DB.

Overall, the observed times for the mutually recursive queries were faster compared to the standard recursive queries. Table 4.1 below contains the times in seconds for standard and mutual recursion at the largest measured input size for each query:

	Force	Late	Margin	Packing	Supply	VM
Standard recursive	172s	1976s	1365s	2011s	912s	1750s
Mutually recursive	150s	1520s	1272s	1885s	879s	1570s
Difference	-12.8%	-23.1%	-6.8%	-6.3%	-3.6%	-10.3%

Table 4.1.: Performance of standard and mutual recursion (in seconds) in Materialize DB

The query `Ship` is excluded because it is non-recursive and lacks `JUMPs` to replace. For the queries `Force` and `Late`, the times shown are not for the largest input sizes, because Materialize DB filled up the RAM and became unresponsive when trying to compute the result for the entire input.

For **Force**, the times in the table were measured with 100 instead of 1000 random bodies, and for **Late**, with 100,000 instead of 729,413 entries of the **orders** table produced by a German supplier.

For all queries that completed successfully, the mutual recursive variant was consistently faster than the standard recursive variant. On average, mutual recursion improved performance by approximately 10.5% over standard recursion for these queries.

When comparing these execution times with PostgreSQL, most queries took significantly longer in Materialize DB, regardless of whether mutual recursion was used. Table 4.2 presents a comparison of the times for each query in both DBMSs. For Materialize DB, the times listed are for the mutually recursive variants of the queries, except for **Ship**, which is not recursive.

	Force	Late	Margin	Packing	Ship	Supply	VM
PostgreSQL	0.75s	503s	352s	426s	32388s	85s	9220s
Materialize DB	150s	1520s	1272s	1885s	45s	879s	1570s
Difference	+19,900%	+202%	+321%	+342%	-99.86%	+934%	-83%

Table 4.2.: Performance (in seconds) in PostgreSQL and Materialize DB

For **Force** and all recursive TPC-H queries, PostgreSQL outperformed Materialize DB on the largest input size for which the computation was completed. For **VM** and **Ship**, PostgreSQL was faster on small input batches but scaled worse than Materialize DB, becoming slower on the largest input size tested. Examining the times for different input batches reveals that Materialize queries have an overhead of several seconds for all tested queries. This overhead is particularly notable for small input sizes as shown in Figure 4.1 for the query **Margin**; the other queries show similar effects.

Both DBMSs take less time per row as the number of rows increases, indicating an overhead for tasks such as planning the query or outputting the results, which are performed regardless of the input size. The larger the input batch, the more time is required for the actual computation, reducing the impact of the overhead on the total time per row. For Materialize DB, the overhead is significantly higher than for PostgreSQL. When input sizes are large, the time needed per batch row seems to approach a constant function for almost all queries, corresponding to a linear time complexity.

The exception is **VM**, which computes the first n entries of the Padovan sequence for a given input number n . Since this operation is superlinear, both PostgreSQL and Materialize show an increase in time per sequence entry for higher values of n . This trend is illustrated in the graph in Figure 4.2. The graphs for the other tested programs are located in the appendix.

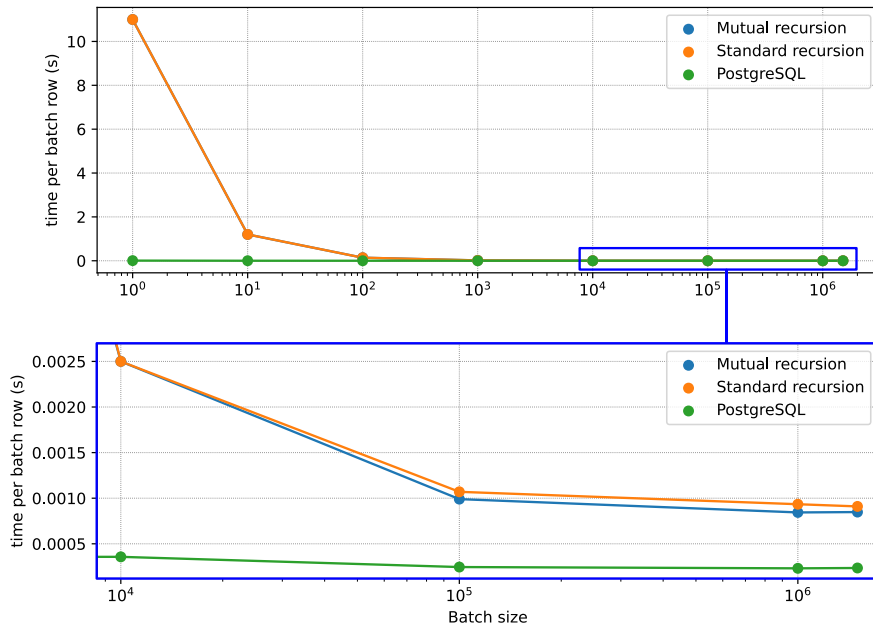


Figure 4.1.: Measured times for Margin across different batch sizes

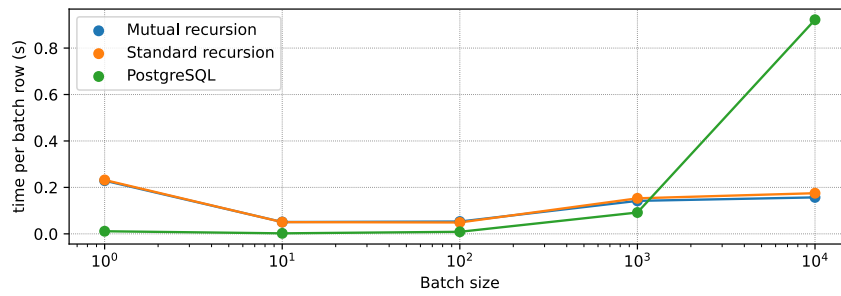


Figure 4.2.: Measured times for VM across different values of n

4.2.2. Performance of Incremental Updates

Incremental updates for a compiled PL\Flummi program were tested as demonstrated in Listing 4.1. The entire compiled query defines a materialized view, which is initially computed by selecting all its values. When any underlying data changes, these changes are propagated to the view using differential dataflow, updating only the affected parts of the computed PL\Flummi query in the materialized view without needing to re-run the entire query for all underlying data rows. This allows to define a complex program flow in PL\Flummi, compile it into a SQL query, and set up a materialized view based on it, ensuring up-to-date results for the current underlying data. We expect this to reduce delay by a lot compared to reevaluating the query over all data rows when only a small part of the input changes.

```

1 CREATE MATERIALIZED VIEW
2 PROGRAM_V(<columns>) AS
3 (
4     <compiled query>
5 );
6
7 -- Compute materialized view
8 SELECT * FROM PROGRAM_V;
9
10 -- Test incrementalization
11 INSERT INTO <underlying data>;
12 SELECT * FROM PROGRAM_V;
```

Listing 4.1: Materialized view from PL\Flummi query

I tested incremental updates with the queries on TPC-H data. For **Late**, the computation for the entire dataset did not complete on the machine, as described above, so the materialized view did not compute either. The results for the other TPC-H queries are shown in Table 4.3:

	Margin	Packing	Ship	Supply
Initial computation	1351s	5039s	15.1s	957s
Update on data	0.48s	2.6s	0.68s	0.34s
Recomputation	7.0s	24s	0.64s	6.0s

Table 4.3.: Performance (in seconds) for incremental updates in Materialize DB

To test incrementalization, I first defined and computed the materialized view, then inserted some rows into an underlying table to alter the computed result for one input of the batch, and recomputed the view. Then I deleted these newly inserted rows and recomputed the view again. Both types of updates (inserts and deletes to an underlying table) always took similar amounts of time to update the materialized view.

For all queries tested, recomputing the materialized view after the underlying data changed took only a fraction of the time required to initially compute it on the entire dataset. Thus, applications for PL\Flummi queries that require up-to-date query results for the current state of data could significantly benefit from incremental updates in materialized views.

4.2.3. Issues with Materialize DB

While conducting performance tests on Materialize DB, two issues arose. As previously mentioned, the DBMS would consume much of the server's RAM, and if the RAM filled up before the query finished, it would not produce any results. This occurred with the **Force** and **Late** queries at the largest input sizes, suggesting that Materialize DB either requires multiple times the size of the data in memory, or may have some memory leak. I have not been able to investigate this further.

Additionally, the Materialize DB daemon process constantly maxed out a single CPU core at 100% load during query evaluation, switching to a different core every 5 to 10 seconds. Although the impact of this behavior on the overall performance of Materialize DB is unclear, repeatedly moving the computation to different cores defeats principles like cache locality, where the data that is frequently needed for the computation is cached within the core to be more quickly accessible. This may be a factor that contributes to generally lower performance of Materialize DB.

4.3. Interpretation

We will review the main results of the benchmarks and how they can be interpreted. The tests consistently showed that mutual recursion is advantageous over standard recursion in Materialize DB. Running PL\Flummi programs on a DBMS that supports mutual recursion and eliminating the need for **JUMPs** appears to be beneficial overall, as it led to a speedup of 3.6% to 23.1% depending on the query tested. We can conclude that naturally following the control flow of **WITH MUTUALLY RECURSIVE** is faster than simulating the same control flow with **JUMPs** and additional instructions.

For incremental updates to materialized views, the tests demonstrated that updating query results after minor changes to the underlying tables can be done in a fraction of the time needed to recompute the entire query. Applications that frequently need updated query results as parts of the underlying data change can significantly benefit from Materialize DB's materialized views.

Overall, Materialize DB proved to be quite slow in the benchmarks. For the recursive TPC-H queries **Late**, **Margin** and **Packing**, Materialize DB took 2 to 4 times longer than PostgreSQL. For **Supply**, Materialize DB took more than 9 times longer. The query on which Materialize DB performed the worst compared to PostgreSQL was **Force**, which involves geometric calculations based on a tree structure and uses arrays (lists in Materialize DB). PostgreSQL has built-in support for many geometric datatypes and operations, whereas I had to manually define these operations in Materialize DB. This likely contributed to the significantly worse performance: Materialize DB took 200 times longer on **Force** than PostgreSQL.

Two queries were faster on Materialize DB than on PostgreSQL: **Ship** and **VM**. **VM** differs from the other queries in that its execution involves many recursive calls on small amounts of data until the n -th Padovan number is computed and the recursion terminates. The only data read is the VM instructions, the main work of the query consists of repeated recursive calls. The TPC-H

queries, as opposed, typically require only few recursive calls per row and run on many input rows. Assuming that there is an overhead associated with using `WITH MUTUALLY RECURSIVE` for many parallel calls with rows from persistent storage, VM's performance advantage on Materialize DB over the TPC-H queries becomes clear.

Regarding the performance of `Ship`, it is important to note that it is the only linear query without loops and, therefore, without recursion. In an exchange with Frank McSherry, co-founder and chief scientist of Materialize Inc., we learned that many optimizations are completely disabled for `WITH MUTUALLY RECURSIVE` queries. This is because the DBMSs cannot easily predict the behavior of the recursive CTE: Rows could be added or removed in each iteration for an unpredictable number of times. This makes many optimizations inapplicable for the query planner. Since `Ship` is the only query that does not use `WITH MUTUALLY RECURSIVE`, it benefits from better query planning and execution optimizations by Materialize DB, explaining its performance advantage over the other queries.

Conversely, the `Ship` query performed poorly in PostgreSQL. Upon investigation, we found that unlike Materialize DB, PostgreSQL does not decorrelate the subqueries that count the modes of transportation for the customers. This leads to an expensive query plan with a lateral join that takes a lot of time. To address this, future enhancements to Flummi's compilation methods could include assigning multiple variables in a single PL\Flummi statement, reducing the correlation of queries, and decreasing the total number of generated CTEs in order to achieve better performance in PostgreSQL.

In our discussions with Frank McSherry, we also uncovered several reasons for the generally suboptimal performance of Materialize DB in our benchmarks. Firstly, Materialize DB is designed as a distributed system, so our single-machine setup is not ideal. Additionally, incrementalization using differential dataflow is one of the key features of Materialize DB, and the DBMS runs the tools required for it even for single queries outside of materialized views. This explains the substantial overhead we observed for small datasets: Materialize DB appears to require this constant amount of time to initialize the execution of complex queries, regardless of whether their state is later tracked in a materialized view.

Other factors contributing to the poor performance include:

- The Flummi compiler introduces casts between `integer` and `bigint` types, which have different hashes. Consequently, Materialize DB sometimes cannot reuse a hash index for casted values.
- Branching is implemented in Flummi using `UNION ALL`s of queries with mutually exclusive `WHERE` predicates. This approach makes it harder for the DBMS to detect that only one of the two queries will produce results compared to using constructs like `CASE WHEN`. Additionally, Materialize DB does not lower the boolean expressions into relational operations (such as `INTERSECT` for `AND`), but evaluates them directly, slowing down query evaluation, especially for large batch sizes.
- Compiled PL\Flummi programs encode all output of a CFG block into the columns of the corresponding CTE. This can result in columns with a constant value across all rows, and more correlated columns than necessary. Materialize DB struggles with decorrelating these

columns, leading to potential performance penalties.

- The TPC-H specification lists one or more columns for each table that uniquely identify a row and can therefore serve as primary keys. Using primary key constraints is generally beneficial for DBMSs, because they help to identify unique rows in queries, for example when **DISTINCT** is used or when an equi-join on tables is performed. Typically DBMSs will also automatically create an index on the primary key of a table to enable faster lookups during key-based filtering, so I introduced all primary and foreign key constraints in PostgreSQL. As mentioned before, the lack of key constraints in Materialize DB only allowed me to manually create indices on the key columns. The absence of actual key constraints for query planning might be a limiting factor for the performance as well.

However, Materialize DB excels in incremental updates of data. In comparison to other DBMSs that need to run the entire query anew each time, the use of materialized views in Materialize DB shows significant advantages. For example, running **Margin** five times in a row on slightly different data in PostgreSQL would take approximately $5 \cdot 352 \text{ s} = 1760 \text{ s}$. In contrast, using a materialized view on Materialize DB and computing a small incremental update four times would only take $1351 \text{ s} + 4 \cdot (0.48 \text{ s} + 7.0 \text{ s}) \approx 1381 \text{ s}$. In such scenarios, which are likely to have real-world applications, Materialize DB is more efficient for the PL\Flummi queries over time than PostgreSQL in the long run.

Overall, while the performance of Materialize DB is rather poor for the reasons discussed above, it also offers advantages that justify considering it as a target platform for Flummi. Mutual recursion proved to be more efficient than standard recursion in all tests, and incrementalization on materialized views can save significant time, making Materialize DB more time efficient than other DBMSs when queries need frequent repetition. The main issue with Materialize DB is its high memory usage, which in some cases completely halted our queries when running them on the entire dataset.

Conclusion

5.1. Summary

This thesis explored the compilation of PL\Flummi programs into mutually recursive SQL queries executable in Materialize DB using its `WITH MUTUALLY RECURSIVE` construct. The primary focus was on assessing the performance benefits of mutual recursion over standard recursion and the effectiveness of incremental updates in materialized views.

The motivation for this research stemmed from the desire to perform complex computations directly within a relational DBMS to reduce data transfer overhead and enhance efficiency. Writing complex programs as an iterative workflow is easier for many developers than crafting recursive SQL queries. Therefore, the Flummi compiler aims to facilitate writing programs for DBMSs in an imperative style and compiling these workflows into purely recursive SQL.

Because the semantics of `WITH MUTUALLY RECURSIVE` closely align with Flummi's compilation rule for loops, I implemented the compilation of PL\Flummi programs to mutually recursive queries and tested their performance in Materialize DB. My findings revealed that leveraging mutual recursion provided a performance boost ranging from 3.6% to 23.1%, indicating that these computations benefit from Materialize DB's semantics that match their control flow.

I also tested the incrementalization feature, a key aspect of Materialize DB, and found it to work well with PL\Flummi programs. Despite the generally low performance of Materialize DB, using incrementalization can lead to a competitive performance in suitable scenarios. When a query runs within a materialized view, computing the incremental update only takes a fraction of the time needed to reevaluate the query on the entire dataset.

Altogether, Materialize DB proves to be a viable compilation target for Flummi, since mutual recursion consistently outperformed standard recursion. Although Materialize DB's performance was relatively slow, it has the potential to outperform other DBMSs in running compiled PL\Flummi queries, depending on the program to run and on the use of incrementalization.

5.2. Future Work

Given the issues encountered with Materialize DB during our tests, there are several areas for future improvement. Enhancements in memory usage and in the use of multiple processor cores could enable the DBMS to handle larger datasets in complex queries. Additionally, reducing the performance overhead for complex queries and optimizing the query planner for recursive queries could lead to a better overall performance for queries in Materialize DB.

For Flummi, future improvements could include avoiding unnecessary casts, generating CTEs with less correlation, and exploring different approaches to branching that can more easily be understood by DBMSs. Such enhancements could help Materialize DB, as well as other DBMSs, to perform better on compiled PL\Flummi programs because they can unlock optimizations that are currently not performed. More enhancements, such as the ability to assign multiple variables in a single PL\Flummi instruction, are already being explored. These changes could reduce the number of correlated subqueries in some programs, improving the performance of PL\Flummi programs across all DBMSs in the future.



Appendix: Measure Charts

Here are the charts for the other performance measurements of PL\Flummi programs in Materialize DB and PostgreSQL besides Figures 4.1 (Margin) and 4.2 (VM). All charts display the time used per input row for an increasing number of rows, up to the maximum. In Figures A.1 (Force) and A.2 (Late), the data for Materialize DB ends at the second largest input size because the queries did not terminate at the respectively largest size. In Figure A.4 (Ship), there is no distinction between mutual and standard recursion as the query is non-recursive.

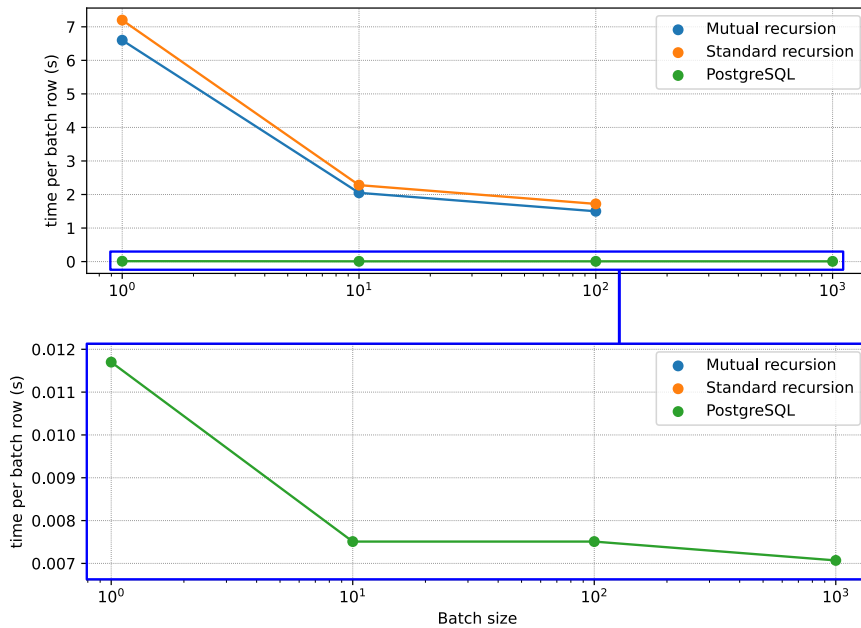


Figure A.1.: Measured times for Force across different batch sizes

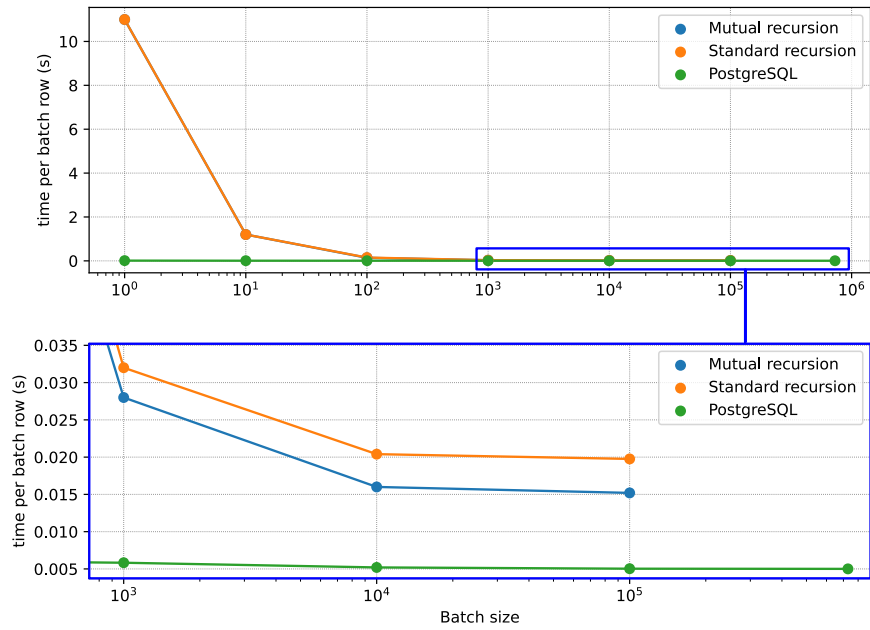


Figure A.2.: Measured times for **Late** across different batch sizes

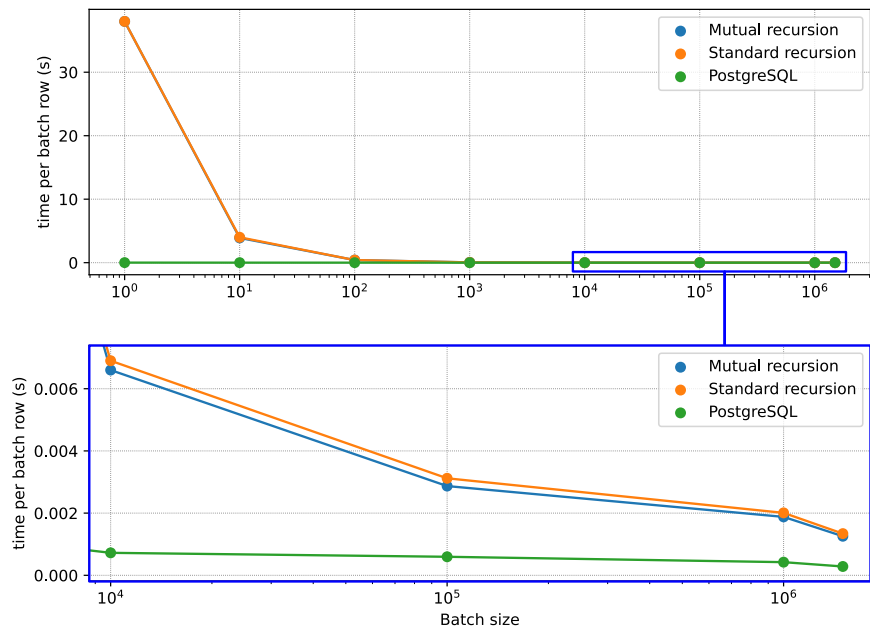


Figure A.3.: Measured times for **Packing** across different batch sizes

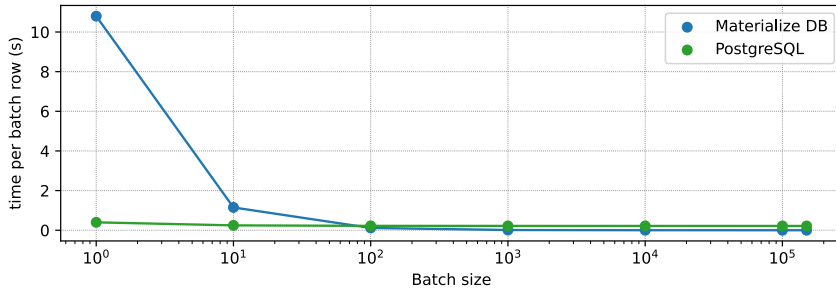


Figure A.4.: Measured times for Ship across different batch sizes

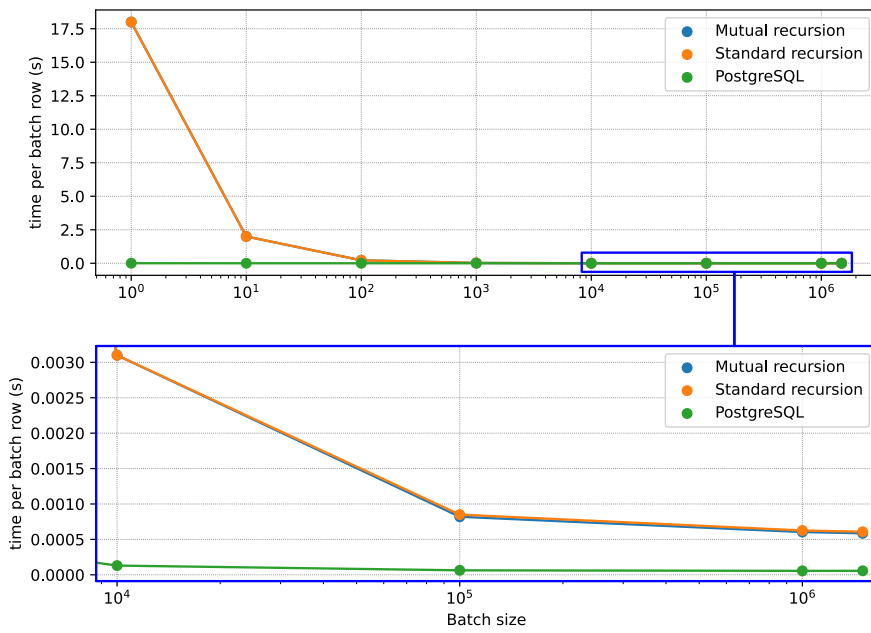


Figure A.5.: Measured times for Supply across different batch sizes

Bibliography

- [1] ISO/IEC 9075-2:1999. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. ISO. 1999.
- [2] Christian Duta. “Another Way to Implement Complex Computations: Functional-Style SQL UDF”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA 2022), collocated with SIGMOD 2022*. Philadelphia, PA, USA, June 2022. URL: <https://db.cs.uni-tuebingen.de/publications/2022/another-way-to-implement-complex-computations-functional-style-sql-udf/fsUDFs-user-study.pdf>.
- [3] Materialize Inc. *The Operational Data Warehouse | Materialize*. <https://materialize.com/>. Accessed: 2024-04-12.
- [4] Tim Fischer, Denis Hirn, and Torsten Grust. *SQL Engines Excel at the Execution of Imperative Programs*. Under submission at VLDB 17. 2024.
- [5] Materialize Inc. *Home | Materialize Documentation*. <https://materialize.com/docs/>. Accessed: 2024-04-12.
- [6] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: 16: SELECT*. <https://www.postgresql.org/docs/current/sql-select.html>. Accessed: 2024-04-18.
- [7] Oracle. *MySQL :: MySQL 8.0 Reference Manual :: 15.2.20 WITH (Common Table Expressions)*. <https://dev.mysql.com/doc/refman/8.0/en/with.html#common-table-expressions-recursive>. Accessed: 2024-04-18.
- [8] Hwaci. *The WITH Clause*. https://www.sqlite.org/lang_with.html. Accessed: 2024-04-18.
- [9] Microsoft. *WITH common_table_expression (Transact-SQL) - SQL Server | Microsoft Learn*. <https://learn.microsoft.com/en-us/sql/t-sql/queries/with-common-table-expression-transact-sql>. Accessed: 2024-04-18.
- [10] Materialize Inc. *Recursive CTEs | Materialize Documentation*. <https://materialize.com/docs/sql/recursive-ctes>. Accessed: 2024-04-02.
- [11] Materialize Inc. *Recursive SQL Queries in Materialize | Materialize*. <https://materialize.com/blog/recursive-ctes-in-materialize/>. Accessed: 2024-04-24.
- [12] Frank McSherry et al. “Differential Dataflow”. In: *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.
- [13] TPC. *TPC-H Homepage*. <https://www.tpc.org/tpch/>. Accessed: 2024-05-31.
- [14] Inc. Materialize. *CREATE SOURCE: Load generator | Materialize Documentation*. <https://materialize.com/docs/sql/create-source/load-generator/>. Accessed: 2024-05-31.
- [15] DuckDB. *TPC-H Extension – DuckDB*. <https://duckdb.org/docs/extensions/tpch.html>. Accessed: 2024-05-31.

- [16] The OEIS Foundation Inc. *A000931* - OEIS. <https://oeis.org/A000931>. Accessed: 2024-05-31.
- [17] Materialize Inc. *Materialize on GitHub*. <https://github.com/MaterializeInc/materialize/tree/e7ac79eaa1>. Accessed: 2024-06-05.