Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Multiple Query Execution through SQL Rewriting

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s): Wolf

First name(s): Jan Thorsten

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

Richterswil, 08.09.2017

Signature(s)

Jan Wolf

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.
Multiple Query Execution through SQL Rewriting

by

Jan Wolf
Legi: 10-922-201

Supervised by

Prof. Dr. Gustavo Alonso
Dr. Darko Makreshanski
Dr. Ingo Müller

September 09, 2017
Abstract

Current database systems typically process queries in a query-at-a-time fashion without considering common subexpressions that can be shared. Such a design leads to suboptimal processing, i.e., lower throughput, higher latency, and poor scalability. Taking advantage of sharing common subexpressions in batches or streams of concurrent queries is one way to mitigate those problems. Another option is to implement new shared operators in database engines, or rewrite them completely. The main aspect of our work is that we achieve shared query execution without modifying any database engines. In this work we present a novel approach that completely relies on SQL query rewriting and already existing features of today’s database engines by minimizing redundant work. We first describe three different methods how to exploit sharing opportunities. Secondly, the methods are evaluated via microbenchmarks on four different common database systems. Our results show that it is very well possible to achieve a significant reduction of execution cost without modifying a database engine. In several cases our methods improve the performance by orders of magnitude.
## Contents

1. **Introduction**

2. **Background and Related Work**
   - 2.1 Overview
   - 2.2 Shared Scan and Shared Join

3. **Multi-Query Execution through SQL Rewriting**
   - 3.1 SQL Rewriting Architecture
   - 3.2 Sharing with Query ID Table
     - 3.2.1 Shared Scan
     - 3.2.2 Shared Join
     - 3.2.3 Shared Aggregation
   - 3.3 Sharing with Arrays
     - 3.3.1 Shared Scan
     - 3.3.2 Shared Join
     - 3.3.3 Shared Aggregation
   - 3.4 Sharing with Bitsets
     - 3.4.1 Shared Scan
     - 3.4.2 Shared Join
     - 3.4.3 Shared Aggregation
     - 3.4.4 The Bitset Datatype
   - 3.5 Putting it together
     - 3.5.1 TPC-H Query 6
     - 3.5.2 TPC-H Query 20

4. **SAP HANA**
   - 4.1 SAP HANA Architecture
   - 4.2 Experimental Environment
     - 4.2.1 Shared Scan Setup
     - 4.2.2 Shared Join Setup
   - 4.3 Query at a Time
   - 4.4 Sharing with Query ID Table
     - 4.4.1 Shared Scan
     - 4.4.2 Shared Join
   - 4.5 Sharing with Arrays
   - 4.6 Sharing with Bitsets
     - 4.6.1 Shared Scan
     - 4.6.2 Shared Join
   - 4.7 Conclusion

5. **SQL Server**
   - 5.1 SQL Server Architecture
   - 5.2 Query at a Time
   - 5.3 Sharing with Query ID Table
     - 5.3.1 Shared Scan
     - 5.3.2 Shared Join
   - 5.4 Sharing with Bitsets
     - 5.4.1 Shared Scan
     - 5.4.2 Shared Join
   - 5.5 Conclusion
1 Introduction

The shift towards cloud computing and service-based IT industry is changing the requirements landscape for database engines. Database systems, in particular for analytical query processing, are increasingly being offered as a service with pay-per-use cost models as opposed to on-premise client-specific installations. For some cases, this results in highly concurrent workloads which are not handled well by existing database engines [13]. The reason is that they process queries one-at-a-time in separate execution contexts, optimizing for low response time of individual queries. Instead, what is required for these highly concurrent workloads is high throughput and guaranteed response time for all queries.

One way of achieving this is to exploit sharing opportunities in concurrent queries, with several techniques being proposed to date. One example is multiple-query optimization (MQO) [11] where common subexpressions of concurrent queries are detected and executed only once. The benefit of this is that existing database engines can be reused, but the drawback is that finding common subexpressions is expensive and it misses opportunities for sharing. Another example is sharing execution of queries with common subplans by annotating intermediate results [3,5,8,13]. This approach exploits more sharing opportunities, reaching higher throughput, however uses a radically different way of query processing requiring a rewrite of the database engine.

This work explores a new way of sharing of query execution, termed Multiple-Query Execution (MQE). The goal is to use a novel query rewriting technique that aims to obtain advantages of both aforementioned approaches, i.e. share execution of queries with common subplans on existing database engines.

The remaining sections are organized as follows: Section 2 discusses related work that inspired certain aspects of MQE. In Section 3 we present a model that shows how MQE can be implemented and explain three different approaches how subexpressions of concurrent queries can be shared. Sections 4 to 7 provide implementation details and analysis of the methods on real database systems. Our main focus lies on OLAP workload on multi-threaded systems. Therefore, the first three database systems use multi-threading and column stores, which are superior to row stores for analytical queries. The fourth system is a single-threaded database system using row stores. The last section, Section 8, concludes the work with a summary of its major findings and future work.

2 Background and Related Work

2.1 Overview

Several methods on multiple-query processing for OLAP workload have been developed in the last decades. MQO [11] initially proposed sharing execution by identifying common subexpression and replacing original subqueries with an improved subquery such that common subexpressions are executed together only once. The drawback is its limited application and the costly detection of common subexpressions. The Volcano optimizer [10] further improves this concept by updating shared, materialized views, whereas the materialization also limits the approach. SharedDB [5] and QPipe [6] use the concept of batch-processing. Whereas SharedDB batches queries during the processing of a previous batch, QPipe batches them during a certain time frame. Both implement a simultaneous pipelining technique that allows queries to reuse previously computed query results, query plans, or materialized views.

Another option is to improve scheduler policies. Zukowski et al. [14] developed the Cooperative Scans framework which performs dynamic scheduling of queries and the requested data to reduce the number of fetched pages. Other systems support the concept of sharing disk bandwidth between queries. Circular scans [6] is such a concept and allows queries to attach
themselves to already running scans. Blink [9] and Crescando [13] are sharing disk and memory bandwidth by processing multiple queries in a single table scan. For example in a scan, a thread usually iterates over each data block for a given query. In Blink, however, a thread takes one data block and processes it for each query which obtains significantly more I/O sharing. In Crescando [13], the authors use also predicate indexes for fast lookup of which attributes belong to which query.

In sensor networks, e.g., for weather forecasting or the Internet of Things, each node frequently generates queries resulting in a large continuous stream of queries. This problem can be tackled by sharing physical operators [7] or grouping queries according to similar structures [4].

2.2 Shared Scan and Shared Join

MQE is partially based on the ideas of MQJoin [8]. The authors introduce a shared join model which adds extra attributes to relations in order to share tuples among queries. For completeness we repeat their definitions of shared scan and shared join here.

Let $R$ and $S$ be two relations, and $t_R \in R$ and $t_S \in S$ be tuples of the corresponding relations. The function $\sigma^R : R \rightarrow \{\top, \bot\}$ defines a select operation on relation $R$. The output of such a scan is denoted as $\sigma^R$. A shared operation operates on a set of queries $Q = \{q_1, q_2, \ldots q_n\}$, where for a shared scan $q_i = \sigma^R_R$ for $i \in \{1, 2, \ldots n\}$, and for a shared join $q_i = \sigma^R_R \bowtie \sigma^S_S$ for $i \in \{1, 2, \ldots n\}$. The result of a shared scan $\sigma^R_Q$ is defined as follows:

**Definition 1:** Shared Scan

$$\sigma^R_Q = \{ (t_R, (b^R_{q_1}, b^R_{q_2}, \ldots b^R_{q_n})) \mid b^R_{q_i} = \top \iff \sigma^R_R(t_R) \land \exists i. b^R_{q_i} = \top \}$$

The shared scan outputs an extended relation. Each tuple has additional Boolean attributes $b^R_{q_i}$, as many as there are queries. Whenever a query is interested in a tuple $t_R$, the corresponding attribute $b^R_{q_i}$ is set to True, i.e. $\sigma^R_R(t_R) = \top$. The final result contains the tuple $t_R$ if at least one query is interested in it, i.e. $\exists i. b^R_{q_i} = \top$. A shared join is then defined as the join between the results of two shared scans $\sigma^R_Q$ and $\sigma^S_S$.

**Definition 2:** Shared Join

$$\sigma^R_Q \bowtie \sigma^S_S = \{ (t_R, t_S, (b^{R\bowtie S}_{q_1}, b^{R\bowtie S}_{q_2}, \ldots b^{R\bowtie S}_{q_n})) \mid b^{R\bowtie S}_{q_i} = \top \iff (b^R_{q_i} = \top \land b^S_{q_i} = \top) \land \exists i. b^{R\bowtie S}_{q_i} = \top \land f_\bowtie(t_R, t_S) \}$$

Where $f_\bowtie : R \times S \rightarrow \{\top, \bot\}$ is the join predicate function and $(t_R, t_S)$ is a concatenation of the attributes $t_R$ and $t_S$. The join predicate function needs to be the same for all queries so that the join can be shared. The shared join outputs an extended relation with an additional Boolean attribute $b^{R\bowtie S}_{q_i}$ for every query as well. In this case, $b^{R\bowtie S}_{q_i}$ is the result of the conjunction of the corresponding attributes of the input relations, i.e. $b^R_{q_i} \land b^S_{q_i}$. This means that when a query is interested in a tuple $t_R$ and a tuple $t_S$ then those tuples get joined according to $f_\bowtie$. Again, the final result contains only tuples where at least one query is interested in.

In MQJoin the Boolean attributes $b_{q_i}$ are represented by a bitset where each bit corresponds to one attribute. This allows efficient computations, e.g. of the conjunction in the shared join, by using bitwise operators and bit-manipulation functions.
This work investigates three possible methods how to exploit sharing opportunities for workloads with high concurrency in existing database systems with plain SQL rewriting. The SQL rewriting process is detailed in Section 3.1. All three presented methods are variations of the MQJoin [8] adapted to SQL. The first and second methods are using query IDs or arrays respectively instead of bitsets. The third method is an SQL implementation of MQJoin [8] with bitsets. Sections 3.2 to 3.4 describe these methods and their implementations. In this section we assume an ideal system which supports all needed functionality.

The sharing of OLAP queries can be roughly separated into three subproblems: (i) shared scan, (ii) shared join and (iii) shared aggregation. The third subproblem can also be seen as extracting query-specific tuples from an intermediate result relation that is shared across queries. These subproblems are discussed for each method individually in the following sections. Note that pseudo-code is used and the actual implementations on a real system might differ. Finally, in Section 3.5 we show how these methods can be applied to more complex queries.

### 3.1 SQL Rewriting Architecture

We envision the following architecture of multi-query execution that is depicted in Fig. 3.1. It consists of three parts: the clients, our middleware, and the DBMS. We reference to a query of type X as a query that has a specific subexpression X that can be shared. So for multiple queries of type X it is possible to share this common subexpression X. As an example, let us assume we have two queries, both computing a scan over a relation Y. The first query computes the sum of an attribute whereas the second query counts the number of tuples. Since both queries have a common subexpression X, i.e., the scan on relation Y, they both are of type X. They only differ in the selection.

On the left side we have \( N \) clients that want to execute queries. A client sends arbitrary queries to the collector module of our middleware. Assume that \( M \) queries in total arrive at the collector module. The collector attaches an ID to each query and maps it to the corresponding client. Other than that, the collector gathers all queries of the same type X from all clients during a small time period, batches them together, and sends them to the query generator.
The query generator, generates a shared version of the queries, only using SQL rewriting, which consists of a shared scan or shared join, and a shared aggregation. The result is then sent to the database system. The database system first computes the shared scan or shared join and then the shared aggregation. The shared aggregation returns a result set for each query. The $M$ result sets are sent to the distributor, which distributes them to the corresponding clients using the ID-to-Client map generated by the collector previously.

This architecture is quite flexible and can be extended in many ways. For example, since MQE does not touch the DBMS it is very well possible to move the middleware to the client side. In case a where the client wants to run several queries of the same type, he can generate the shared version by himself and send it directly to the DBMS. In such a scenario the collector and distributor are redundant and can be removed.

### 3.2 Sharing with Query ID Table

The method explained here uses a query ID table, called $QIDTABLE$, to annotate tuples with query IDs of interested queries. The idea is to duplicate each tuple for every query that is interested in it, which is achieved by a cross join. This might seem to be an awful idea for full table scans but we expect to gain performance boosts for very selective queries or for relations with few tuples.

#### 3.2.1 Shared Scan

The general implementation of this approach for $N$ scan queries on a table $X$ is shown in Listing 3.1. The filter $QID \leq N$ is needed to tell the optimizer not to use the whole $QIDTABLE$ for the cross join but only the necessary IDs. Another possibility is to create the $QIDTABLE$ for every batch with the exact number of queries. The process of the duplication for a shared scan with two queries is explained in Example 1.

**Listing 3.1: Shared scan with QIDTABLE**

```sql
1 SELECT *
2 FROM X, QIDTABLE
3 WHERE QID \leq N AND
4   ( ( <QUERY_1_PREDICATES> AND QID = 1) OR
5     ( <QUERY_2_PREDICATES> AND QID = 2) OR
6       ...
7     ( <QUERY_N_PREDICATES> AND QID = N) )
```

**Example 1:** Sample Shared Scan of table $X$ for two queries.

Assume queries $Q1$ and $Q2$ and tables $X$ and $QIDTABLE$ as below. The shared scan is computed by executing a cross join between those tables. Afterwards a filter with the predicates of $Q1$ and $Q2$ is applied. This results in a table of tuples from the scanned table combined with query IDs. Each tuple is duplicated for every query whose predicate it fulfills. In this case, since the second tuple (XID = 2) fulfills the predicates for $Q1$ as well as $Q2$ it appears twice in the result table, once with query ID 1 and once with query ID 2.
3.2.2 Shared Join

This section explains how the shared join between two tables \( X \) and \( Y \) is implemented in pure SQL. Assuming \( N \) queries that do an equi-join on columns \( X.A \) and \( Y.B \) as shown in Listing 3.2. The pseudo-code is listed in Listing 3.3. In a first step, the shared scan is computed on both tables, here represented by intermediate results \( i_X \) and \( i_Y \). For this step it is necessary to figure out the predicates for all queries of the corresponding tables. The second and already last step is to do a natural join between the intermediate result tables on their \( QID \) attributes. The whole process is illustrated again in Example 2.

**Listing 3.2: Query with range scan predicate on column X_ID**

```sql
1 SELECT *
2 FROM X JOIN Y ON X.A = Y.B
3 WHERE (<QUERY_PREDICATES_ON_X>) AND
4   ( <QUERY_PREDICATES_ON_Y> );
```

**Listing 3.3: Shared join implementation using intermediate results**

```sql
1 BEGIN
2   i_X = SELECT * FROM SharedScan(X, QIDTABLE);
3   i_Y = SELECT * FROM SharedScan(Y, QIDTABLE);
4
5   SELECT *
6     FROM i_X JOIN i_Y
7     ON i_X.A = i_Y.B
8     AND i_X.QID = i_Y.QID;
9 END;
```

**Example 2:** Sample Shared Join between tables \( X \) and \( Y \) for three queries.

Assume queries \( Q1 \), \( Q2 \) and \( Q3 \), all of the same type of query, i.e. an equi-join on attributes \( X.A \) and \( Y.B \) with predicates on \( XID \) and \( YID \). Only the filter values differ.
i_X and i_Y are intermediate relations produced by shared scans as below. The shared join is computed by executing an equi-join between the intermediate tables on attributes i_X.A and i_Y.B as well as on the QID attributes. The final result table of the shared join contains 4 tuples two of which belong to query 1, two to query 2 and none to query 3.

Q1: 
```sql
SELECT *
FROM X JOIN Y ON X.A = Y.B
WHERE ( XID BETWEEN 1 AND 2 ) AND ( YID = 4 );
```

Q2: 
```sql
SELECT *
FROM X JOIN Y ON X.A = Y.B
WHERE ( XID BETWEEN 2 AND 3 ) AND ( YID BETWEEN 1 AND 2 );
```

Q3: 
```sql
SELECT *
FROM X JOIN Y ON X.A = Y.B
WHERE ( XID = 4 ) AND ( YID = 4 );
```

3.2.3 Shared Aggregation

The last step is to extract query-specific tuples from a shared result relation. This is where the advantage of this method comes into play. Since every tuple is matched with a single query ID, aggregate the tuples by this ID allows us to fetch the intended tuples. Assume table R is the result of a shared scan or shared join from the previous sections, Listing 3.4 shows the SQL implementation of a shared aggregation for a single query. The result set of this statement consists of the tuples requested by the query.

Listing 3.4: Collecting tuples belonging to query i.
```sql
1 SELECT *
2 FROM R
3 WHERE QID = i ;
```

In the same way we handle queries with operators that are applied after the select clause has been executed. Common operators are aggregate functions, such as MAX or COUNT, and SQL-specific clauses, such as GROUP BY, ORDER BY or LIMIT. Listing 3.5 shows the SQL implementation of a shared aggregation for a query with an ORDER BY and a LIMIT clause.

Listing 3.5: Shared aggregation for a query with an ORDER BY and a LIMIT clause.
```sql
1 SELECT *
2 FROM R
3 WHERE QID = i
4 ORDER BY <QUERY_i_ORDER_ATTRIBUTE>
5 LIMIT <QUERY_i_LIMIT> ;
```
3.3 Sharing with Arrays

The method explained here is a modified MQJoin [8] that uses arrays instead of bitsets. The advantage of arrays is that a single array can be used to store thousands of query IDs. This reduces the overhead of processing duplicated tuples like in Section 3.2.

The disadvantage, on the other hand, is that they are not SQL standard but database-specific data types, thus not available in all database systems. Another problem is that often the needed operations and features are missing, such as intersection of arrays.

3.3.1 Shared Scan

First, we show how the shared scan can be implemented with arrays. Afterwards, we demonstrate with an example how this approach can be improved in certain cases by applying a binary search algorithm.

**Inline.** Listing 3.6 displays pseudo-code how the shared scan is inlined in SQL. Each element of the array corresponds to a query ID and contains an expression that evaluates the predicate of the query. If the expression evaluates to true then the element gets set to the query’s ID, otherwise to 0. The function ARRAY_REMOVE takes as input the array and a number and removes all occurrences of that number from the array. In this case all zeros are removed such that only valid query IDs remain in the array. This results in an extended table X where each tuple has an additional column \( QIDS \) containing an array with all query IDs that are interested in the tuple.

**Listing 3.6: Shared scan with arrays using inline implementation**

<table>
<thead>
<tr>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

**Binary Search Inline.** The inline approach can be improved by applying a binary search algorithm which precomputes the arrays for each predicate value. This is related to predicate indexes as used in Crescendo [13]. Since we rely only on SQL rewriting, we cannot implement our own index structure. Therefore we came up with the binary search trick. However, this trick is only applicable for queries with predicates on a single attribute. The binary search algorithm itself is not trivial and gets very complicated depending on the predicates of the queries. In Example 3 we demonstrate the binary search algorithm for range predicates on a single attribute.

**Example 3:** Binary search algorithm for range predicates on a single attribute.

The following example explains how the binary search algorithm looks like for range predicates. Assume four queries as shown below with range bounds \( L_i \) and \( U_i \) from Table 3.1.
Query i: \( \text{SELECT} \ast \text{FROM X WHERE XID BETWEEN } L_i \text{ AND } U_i; \)

Table 3.1: Predicate bounds for four queries.

<table>
<thead>
<tr>
<th>Query (i)</th>
<th>Lower Bound ((L_i))</th>
<th>Upper bound ((U_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Fig. 3.2 illustrates how the binary search computes the query ID arrays. At the bottom we see the values of the predicate attribute \(XID\). The black lines in the middle indicate the predicate ranges for each query. At the top are the computed query IDs that are returned by the binary search as arrays. Each array corresponds to a distinct section of \(XID\) values.

Array

![Array Diagram](image)

Figure 3.2: Computation of the arrays for the binary search.

In a last step the algorithm applies a binary search on the previously found sections and returns the corresponding arrays. This results in SQL code as shown in Listing 3.7. With this, the arrays are precomputed and returned directly for each predicate. In cases where no predicate is fulfilled the array is set to NULL.
Listing 3.7: Shared scan with arrays using binary search.

```sql
SELECT *
CASE WHEN XID <= 6 THEN
  CASE WHEN XID <= 3 THEN
    CASE WHEN XID = 1 THEN ARRAY(1)
    WHEN XID >= 2 AND XID <= 3 THEN ARRAY(1,2)
  END
  ELSE
    CASE WHEN XID = 4 THEN ARRAY(1,2,3)
    WHEN XID >= 5 AND XID <= 6 THEN ARRAY(1,3)
  END
END
ELSE
  CASE WHEN XID <= 9 THEN
    CASE WHEN XID = 7 THEN ARRAY(3)
    WHEN XID >= 8 AND XID <= 9 THEN ARRAY(4)
  END
END
ELSE
  CASE WHEN XID <= 6 THEN
    CASE WHEN XID <= 3 THEN
      CASE WHEN XID = 1 THEN ARRAY(1)
      WHEN XID >= 2 AND XID <= 3 THEN ARRAY(1,2)
    END
    ELSE
      CASE WHEN XID = 4 THEN ARRAY(1,2,3)
      WHEN XID >= 5 AND XID <= 6 THEN ARRAY(1,3)
    END
  END
ELSE
  CASE WHEN XID <= 9 THEN
    CASE WHEN XID = 7 THEN ARRAY(3)
    WHEN XID >= 8 AND XID <= 9 THEN ARRAY(4)
  END
END
AS qids
FROM X;
```

Shared Scan with Predicate Filter. At this state the whole relation gets scanned and the arrays are computed for every single tuple, most of them just filled with zeros. The shared scan can be further improved by simply applying a pre-filter with the conjunction of the query predicates to it. For high selective queries this greatly reduces the amount of tuples that need to be processed, and therefore also reduces the amounts of arrays that are generated. This, of course, has a positive impact on the shared join.

### 3.3.2 Shared Join

This section shows how the shared join is implemented. Assuming \( N \) queries with an equi-join on columns \( X.A \) and \( Y.B \), same as for the \textit{QIDTABLE} approach in Listing 3.2. The desired implementation for a shared join would look like Listing 3.8. In a first step we compute the shared scan on tables \( X \) and \( Y \) and store them in intermediate results \( i_X \) and \( i_Y \).

With the array intersection we figure out which tuples are still interesting after the join. If the array is empty after the join, then no query is interested in the tuple. The function \texttt{ARRAY\_EMPTY} checks if the array is empty in order to filter out unused tuples.

Listing 3.8: Shared join implementation using intermediate results

```sql
BEGIN
i_X = SELECT * FROM SharedScan(X) WHERE qids NOT NULL;
i_Y = SELECT * FROM SharedScan(Y) WHERE qids NOT NULL;

SELECT *, final_qids
FROM (SELECT ARRAY\_INTERSECT(i_X.qids , i_Y.qids) AS final_qids
  FROM (i_X JOIN i_Y ON i_X.A = i_Y.B)
) WHERE NOT ARRAY\_EMPTY(final_qids);
END;
```
3.3.3 Shared Aggregation

Since the arrays already contain the query IDs per tuple, all that needs to be done is to compute the cross product between the tuple and the array. Afterwards we can filter the result by a query ID. The latter is the same as described in Section 3.2.3.

For the cross product two tables are needed. This means the arrays need to be transformed to tables with a single column which can be achieved with an UNNEST function. Assume that $R$ is a table with a $QIDS$ column as depicted in Table 3.2. The desired implementation is shown in Listing 3.9.

Table 3.2: Result table $R$ of a shared join (left), $R$ after the cross product with the $QIDs$ (middle) and $R$ after the group by $QID$ (right)

<table>
<thead>
<tr>
<th>Col</th>
<th>QIDS</th>
<th>Col</th>
<th>QID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 3, 4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2, 3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4, 5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Listing 3.9: Extracting tuples of query $i$ from the result table $R$ of a shared scan or shared join.

1. SELECT *
2. FROM R CROSS JOIN UNNEST(QID)
3. WHERE QID = i;

3.4 Sharing with Bitsets

As described in Section 2 MQJoin [8] is able to share tuples among queries with a shared scan and shared join approach using bitsets. This method introduces a challenge that needs to be tackled for existing systems, how to store the bitsets. The size of the bitsets depend on the number of queries in the processed batch. Each query is related to a single bit in the bitset, therefore, processing $N$ queries requires bitsets of size $N$. Sections 3.4.1 to 3.4.3 explain possible implementations for the shared scan, the shared join and how to extract query-specific tuples from the shared result relation. Section 3.4.4 discusses different approaches on how to store the bitsets, i.e. the right data type choice for bitsets.

3.4.1 Shared Scan

This section lists two possible approaches how to implement the shared scan with bitsets: (i) inlining and (ii) User Defined Functions (UDF).

Inlined. Assuming $N$ queries with predicates on table $X$. For every query the predicates gets evaluated to 1 if the predicates are fulfilled, otherwise to 0. The result is then multiplied by the corresponding bit value of the query, i.e. $2^id$. The sum of all those bit values results in the bitset indicating which queries are interested in the tuple.

This can be achieved in pure SQL with CASE expressions and is presented in Listing 3.10. The interesting part is that it can be achieved by mere rewriting of SQL queries. Hence, this implementation is pure SQL and can be used on all systems.
Listing 3.10: Shared scan inline implementation.

```sql
SELECT (CASE WHEN <QUERY_0_PREDICATES> THEN 0 ELSE 0 END)
+ (CASE WHEN <QUERY_1_PREDICATES> THEN 1 ELSE 0 END)
+ ... 
+ (CASE WHEN <QUERY_N_PREDICATES> THEN N ELSE 0 END)
AS bitset
FROM X;
```

**Binary Search Inline.** The inline approach can be improved by applying a binary search algorithm, similar as explained in Section 3.3.1. The only difference is that now, instead of arrays, the bitset are precomputed. Assuming $N$ queries with query IDs ranging from 0 to $N-1$, the bitsets for each section are computed as follows:

$$Bitset = \sum_{i=0}^{N-1} 2^{id}$$

Executing the algorithm with the same four queries as in Example 3 results in Listing 3.11.

**User Defined Function (UDF).** Instead of inlining the expressions it is possible to use UDFs to extract complex calculations. Listing 3.12 shows how the bitset is computed with an UDF. The predicates have to be syntactical transformed into the corresponding SQL script language that supports UDFs. The disadvantage is that UDFs are not SQL but database specific. So whether UDFs are supported at all or support all the needed features varies from system to system.
Shared Scan with Predicate Filter. As for arrays, the whole relation gets scanned and the bitsets are computed for every single tuple, most of them being zero. The shared scan can be further improved by simply applying a pre-filter with the conjunction of the query predicates to it. For high selective queries this greatly reduces the amount of tuples that need to be processed, and therefore also reduces the amounts of bitsets that are generated. This, again, has a positive impact on the shared join.

3.4.2 Shared Join

This section shows how the shared join is implemented in pure SQL. Assuming relations $X$, $Y$ and $N$ queries with an equi-join on columns $X.A$ and $Y.B$, same as in Listing 3.2. The desired implementation for a shared join looks like Listing 3.13. The implementation is similar to the one from Section 3.3.2. In a first step the shared scans of the involved relations are computed and store in intermediate results $i_X$ and $i_Y$. Instead of the array intersection we now apply the BITWISE_AND function. Any tuple whose final bitset is not zero has at least one query that is interested in it. Any other tuples are filtered out with the last filter.

Listing 3.13: Shared join implementation using intermediate results

```
BEGIN
  i_X = SELECT * FROM SharedScan(X) WHERE bitset <> 0;
  i_Y = SELECT * FROM SharedScan(Y) WHERE bitset <> 0;
  SELECT *, final_bitset
  FROM (SELECT BITWISE_AND(i_X.bitset, i_Y.bitset) AS final_bitset
        FROM (i_X JOIN i_Y ON i_X.A = i_Y.B)
        )
  WHERE final_bitset <> 0;
END;
```

3.4.3 Shared Aggregation

The problematic part of using bitsets in existing systems is how to extract query-specific tuples from a shared result relation. The biggest limitation is that it is not possible to use aggregation with bitsets, e.g. to group by specific bit values. The authors of the original MQJoin [8] use the Count Leading Zeros (CLZ) functionality to figure out which bits are set to 1. Such a function does not exist in current database systems.

To achieve this in pure SQL a query ID table is necessary. An $QIDTABLE$, such as used in Section 3.2 extended with a bitset column can accomplish this. For query ID $i$ the $ith$ bit is set to
1 in the bitset. Example 4 illustrates the approach in more detail. For a general implementation see Listing 3.14. Two other options would be to have either a CLZ or LOG UDF that can be used to find bits that are set to 1, but we will not go into further details on these options.

Example 4: Sample of extracting tuples per query with bitsets.

Assume a sample result table $R$ after a shared scan or shared join. $R$ is natural joined with $QIDTABLE$ on the bitset attribute to produce a similar output as with the original query ID table method. The extraction of tuples happens the same as explained in Section 3.2.3.

<table>
<thead>
<tr>
<th>R</th>
<th>Bitset</th>
<th>QIDTABLE</th>
<th>Bitset</th>
<th>Final Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RID</td>
<td>Bitset</td>
<td>QID</td>
<td>Bitset</td>
<td>RID</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1</td>
<td>0001</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1110</td>
<td>2</td>
<td>0010</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>0010</td>
<td>3</td>
<td>0100</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4</td>
<td>1000</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Listing 3.14: Extracting tuples for query $i$ from a shared result relation $R$.

```sql
1 SELECT *
2 FROM ( 
3     SELECT R.*, QID
4     FROM R, QIDTABLE as Q
5     WHERE BITWISE_AND(R.bitset, Q.bitset) <> 0
6 )
7 WHERE QID = i;
```

3.4.4 The Bitset Datatype

In SQL there is no native bitset data type. However, there are other data types that can be used to simulate a bitset. The following list discusses data types that come into consideration:

- **Binaries**: Binary types are used to store bytes of binary data can store up to thousands of bytes. Hence, the size of the data type is perfect to process large number of queries. The main issue is that binary types are by far not optimized for bitwise operations rendering these types very unattractive for the purpose.

- **Integers**: Integers are typically limited to specific number of bytes. Every system has a largest integer type of $B$ bits which allows to store up to $B$ queries (if it is unsigned). To simulate a bitset of size larger than the limit $B$, multiple integers are needed.

The only working data type, in terms of efficiency, therefore are integer types. So when dealing with many queries multiple integers are necessary to simulate the bitsets. For further discussion the largest integer is assumed to be $B$ bits (referred to as `BIGINT`). Also it is assumed that all $B$ bits can be used
to store $B$ queries, even though on most systems the integers are signed data types, i.e. can effectively store only $(B - 1)$ queries.

Table 3.3 depicts in more detail how a single bitset can be simulated with multiple integers. Assume number of queries to be $N$ and the largest integer type is $B$ bits. Note that when $N < B$ the upper half ($Bitset2$) is not needed, i.e. only one integer is used.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$B$</th>
<th>Max Value</th>
<th>Single Bitset</th>
<th>Multiple Integers ($B$ bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>5</td>
<td>na 0101</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>255</td>
<td>97</td>
<td>0110 0001</td>
</tr>
</tbody>
</table>

The fact of having limited size of bits affects the discussed implementations of the shared scans and shared joins when $N > B$. The major changes on the implementations are listed below.

**Shared Scan (Inline).** For $N = 2 \times B$ the inline implementation now computes two bitsets with two separate sums of `CASE` expressions as in Listing 3.15. The same holds for the binary search approach.

```
SELECT (CASE WHEN <QUERY_0_PREDICATES> THEN 0 ELSE 0 END)
+ (CASE WHEN <QUERY_1_PREDICATES> THEN 1 ELSE 0 END)
+ (CASE WHEN <QUERY_{B−1}_PREDICATES> THEN 2^{B−1} ELSE 0 END)
AS bitset1 ,
(CASE WHEN <QUERY_B_PREDICATES> THEN 0 ELSE 0 END)
+ (CASE WHEN <QUERY_{B+1}_PREDICATES> THEN 1 ELSE 0 END)
+ (CASE WHEN <QUERY_{N−1}_PREDICATES> THEN 2^{B−1} ELSE 0 END)
AS bitset2
FROM X;
```

**Shared Scan (UDF).** The UDF approach computes a single bitset, if $N > B$ multiple such UDFs are necessary. E.g. for $N = 2 \times B$ two different UDFs are needed to compute the two bitsets:

```
SELECT BITSET1(<PREDICATE_ATTRIBUTE>) AS bitset1 ,
BITSET2(<PREDICATE_ATTRIBUTE>) AS bitset2
FROM X;
```

This can be improved by computing all bitsets at once with a single UDF. Listing 3.17 displays the pseudo-code for $N = 2 \times B$. 

---

14
Listing 3.17: Shared scan implementation using a single UDF to compute two bitsets.

```sql
CREATE FUNCTION BITSETS(val <TYPE>) RETURNS bitset0, bitset1 BIGINT AS BEGIN
    bitset0 = 0;
    if ("QUERY_0_PREDICATES") then
        bitset0 = bitset0 + 20;
    end if;
    ...;
    if ("QUERY_{B-1}_PREDICATES") then
        bitset0 = bitset0 + 2^B-1;
    end if;
    bitset1 = 0;
    if ("QUERY_B_PREDICATES") then
        bitset1 = bitset1 + 20;
    end if;
    ...;
    if ("QUERY_{N-1}_PREDICATES") then
        bitset1 = bitset1 + 2^N-1;
    end if;
    END;
SELECT BITSETS("PREDICATE_ATTRIBUTE") AS (bitset0, bitset1) FROM X;
```

**Shared Join.** The shared join is affected by the bitset size as well. Listing 3.18 shows a shared join implementation with two bitset columns, e.g. in a case where N = 2 * B. The bitset operations are now duplicated for every additional bitset, introducing a constant overhead for every B additional queries.

Listing 3.18: Shared join implementation using intermediate results

```sql
BEGIN
    i_X = SELECT * FROM SharedScan(X)
    WHERE (bitset0 <> 0 or bitset1 <> 0);
    i_Y = SELECT * FROM SharedScan(Y)
    WHERE (bitset0 <> 0 or bitset1 <> 0);
    SELECT *, final_bitset_0, final_bitset_1
    FROM (SELECT BITWISE_AND(i_X.bitset_0, i_Y.bitset_0) AS final_bitset_0,
            BITWISE_AND(i_X.bitset_1, i_Y.bitset_1) AS final_bitset_1
    FROM (i_X JOIN i_Y ON i_X.A = i_Y.B)
    )
    WHERE final_bitset_0 <> 0 or final_bitset_1 <> 0;
END;
```

3.5 Putting it together

In this section we look at more complex queries that are expected in business systems. We decided to use the TPC-H Benchmark [2] for that purpose. TPC-H is a decision support benchmark that comes with a database tables generator and a set of business-oriented ad-hoc queries. The generated data and queries are designed with broad industry-wide relevance. TPC-H has a total of 22 decision support queries. We look at two such queries and rewrite them according to the previously presented methods. Even though it is possible to combine our methods into hybrids, we focus on each method individually. For each TPC-H query, we assume to have two concurrent queries. Query specific predicates or parameters are prefixed with Q1 and Q2 for the first and second query respectively.
3.5.1 TPC-H Query 6

Our first query is Q6, which is shown in Listing 3.19. The query has predicates on three different attributes and expects three parameters Q\_DATE, Q\_DISCOUNT, and Q\_QUANTITY. Listing 3.20 is the shared inline version of Q6 using bitsets and Listing 3.21 using arrays. As mentioned in Section 3.3.1 the binary search algorithm can only be used with single attributes, hence, in order to apply it we need to decide which attribute to pick. The natural choice would be the one that is the most selective. However, for simplicity and ease of presentation, we only discuss the standard inline approach.

In both versions, the common expressions are combined and executed together in a single query. First the bitsets or arrays are computed and empty ones get removed. In a next step the join with the query IDs is computed and in a last step the shared aggregation gets computed by grouping by qid.


```
select
  sum(l\_extendedprice \* l\_discount) as revenue
from
  lineitem
where
  l\_shipdate >= date ' [Q\_DATE] ' 
  and l\_shipdate < date ' [Q\_DATE] ' + interval '1' year 
  and l\_discount between [Q\_DISCOUNT] - 0.01 and [Q\_DISCOUNT] + 0.01 
  and l\_quantity < [Q\_QUANTITY];
```

Listing 3.20: Shared TPC-H Query 6 using bitsets.

```
select
  sum(l\_extendedprice \* l\_discount) as revenue, qid
from
  ( subquery for the scan: removes empty bitsets
    select *
    from
      ( one case clause with all predicates for each query
        ( lines 6–9 in original query)
        case when l\_shipdate >= date ' [Q1\_DATE] ' 
          and l\_shipdate < date ' [Q1\_DATE] ' + interval '1' year 
          and l\_discount between [Q1\_DISCOUNT] - 0.01 
          and [Q1\_DISCOUNT] + 0.01 
          and l\_quantity < [Q1\_QUANTITY] 
        then 1 else 0 end + 
        case when l\_shipdate >= date ' [Q2\_DATE] ' 
          and l\_shipdate < date ' [Q2\_DATE] ' + interval '1' year 
          and l\_discount between [Q2\_DISCOUNT] - 0.01 
          and [Q2\_DISCOUNT] + 0.01 
          and l\_quantity < [Q2\_QUANTITY] 
        then 2 else 0 end
      ) as bitset
    from
      lineitem
  ) SHARED\_TABLE join QIDTABLE — unpack bitsets into QIDs
    on BITWISE\_AND(bitset, QIDTABLE.bitset) <> 0
  group by
    qid; — separate results of different queries
```
Listing 3.21: Shared TPC-H Query 6 using arrays.

```sql
select
  sum(l_extendedprice * l_discount) as revenue, qid
from
  (--- subquery for the scan: removes empty arrays
    select *
    from
      (--- one case clause with all predicates for each query
        (lines 6-9 in original query)
        case when l_shipdate >= date '[Q1_DATE]' 
          and l_shipdate < date '[Q1_DATE]' + interval '1' year 
          and l_discount between [Q1_DISCOUNT] - 0.01 
          and [Q1_DISCOUNT] + 0.01 
          and l_quantity < [Q1_QUANTITY]
        then 1 else 0 end ,
        case when l_shipdate >= date '[Q2_DATE]' 
          and l_shipdate < date '[Q2_DATE]' + interval '1' year 
          and l_discount between [Q2_DISCOUNT] - 0.01 
          and [Q2_DISCOUNT] + 0.01 
          and l_quantity < [Q2_QUANTITY]
        then 2 else 0 end
      ) as qids
    from
      lineitem
  )
where
  cardinality(qids) > 0
) SHARED_TABLE cross join UNNEST(qids) as T(qid)
  --- unpack arrays into single QIDs
  group by
    qid;  --- separate results of different queries
```

Note that for Q6 the shared aggregations differ from the ones presented in Sections 3.3.3 and 3.4.3. The reason is simply because Q6’s select clause consists solely of an aggregation. If that were not the case or the query would contain a `LIMIT` or `ORDER BY` clause, then we would store the shared query (SHARED_TABLE) in an intermediate result which gets materialized. With this we can fetch the desired information from the intermediate result for each query individually.

### 3.5.2 TPC-H Query 20

As the second query we pick Q20, Listing 3.22, a more complex query than Q6. The query has predicates on three different attributes and expects three parameters Q_COLOR, Q_DATE, and Q_NATION. Two issues make this query complex. The first issue is the `IN` condition in line 6. If we applied a method to it as is, we would not be able to associate query IDs of predicates in the inner subquery (Q_COLOR, Q_DATE) with the query ID of the predicate in the outer query (Q_NATION). This means that when we have computed the inner subquery and executed the `IN` condition, we cannot tell which S_SUPPKEY corresponds to which query anymore. The reason for that is because the condition is evaluated only once for all queries together but we need it to get evaluated for each query individually. The same holds for the `IN` condition in line 12. For a better understanding how the rewriting works we explain the rewriting of the inner `IN` condition (lines 7-16 of the original query) in Appendix A.1. The second issue is the `SUM` aggregation within the inner subquery in line 20. The problem is similar as for the `IN` condition. We need to compute the sum for each query individually. Therefore we first have to rewrite the query such that we can evaluate the `IN` condition and compute the sum separately for every query.
For this query we demonstrate the query ID table method in Listing 3.23. If we consider the above issues and put everything together the shared version results in Listing 3.23. The IN conditions have been replaced with implicit joins and by using the query ID table method we can compute the sum aggregation on a per-query basis.


```
select s_name, s_address
from supplier, nation
where s_suppkey in (select ps_suppkey from partsupp
        where ps_partkey in (select p_partkey from part
            where p_name like '[Q_COLOR]%'
        )
        and ps_availqty > (select 0.5 * sum(l_quantity) from lineitem
            where l_partkey = ps_partkey
            and l_suppkey = ps_suppkey
            and l_shipdate >= date(' [Q_DATE] ')
            and l_shipdate < date(' [Q_DATE] ') + interval '1' year
        )
    )
    and s_nationkey = n_nationkey
    and n_name = ' [Q_NATION] '
order by s_name;
```
Listing 3.23: Shared TPC-H Query 20 using a QIDTABLE.

```sql
shared_result =

select
  s_name, s_address, qid
from
  supplier, nation, qidtable,
  ( -- new subquery that replaces the IN conditions
    select
      qid as psqid, ps_suppkey
    from
      partsupp, part, qidtable
    where
      qid <= 2
      and ( -- predicates evaluated for each query individually
        (p_name like '[Q1_COLOR]%  and qid = 1) or
        (p_name like '[Q2_COLOR]%  and qid = 2)
      )
      and ps_partkey = p_partkey -- comes from the inner IN condition
      and ps_availqty > ( select 0.5 * sum(l_quantity)
        from lineitem
        where
          l_partkey = ps_partkey
          and l_suppkey = ps_suppkey
          and ( -- predicates evaluated for each query individually
            l_shipdate >= date('[Q1_DATE]')
            and l_shipdate < date('[Q1_DATE]') + interval '1' year
            and qid = 1)
          or
          (l_shipdate >= date('[Q2_DATE]')
            and l_shipdate < date('[Q2_DATE]') + interval '1' year
            and qid = 2)
        )
      )
    group by
      ps_suppkey, qid
  )

where
  qid <= 2
  and s_nationkey = n_nationkey
  and ( -- predicates evaluated for each query individually
    (n_name = '[Q1_NATION]' and qid = 1) or
    (n_name = '[Q2_NATION]' and qid = 2)
  )
  and psqid = qid
  and ps_suppkey = s_suppkey; -- comes from the outer IN condition

-- extract results for query 1
select s_name, s_address
from shared_result
where QID = 1
order by s_name;

-- extract results for query 2
select s_name, s_address
from shared_result
where QID = 2
order by s_name;
```
4 SAP HANA

In the previous section, the pseudo-code was written under the assumption of an ideal system that supports all needed features. From here on we look at concrete systems. In this section we examine the different sharing methods on the SAP HANA database. SAP HANA is an in-memory database and provides row storage as well as column storage. We consider column storage as it performs better for analytical tasks.

The rest of this section is structured as follows: Section 4.1 gives some basic insights into the architecture of SAP HANA. The experimental setup is defined in Section 4.2 including specification of hardware. In third section, Section 4.3, we analyze the query at a time (QAT) baseline experiments. Sections 4.4 to 4.6 evaluate how the different methods introduced in Section 3 perform compared to the usual QAT processing. In the microbenchmarks we evaluate the performance for different relevant parameters, such as number of concurrent queries, workload size and workload type. In the last section, Section 4.7, we conclude the evaluation for SAP HANA.

4.1 SAP HANA Architecture

We use SAP HANA version 2.0 SP 00 and the command line tool hdbsql for communication. The following characteristics and restrictions of the SAP HANA architecture are relevant for MQE.

SAP HANA uses dictionary compression on all columns by default. Dictionary compression on predicate columns influences the final performance which is an undesired effect. It cannot be disabled but the benefit can be mitigated by using unique values.

It further compresses columns based on the data and data type using different compression methods, therefore, additional compression has been disabled.

SAP HANA does column-at-a-time and operator-at-a-time processing. When the database executor executes the physical execution plan the performance for large expressions and statements decreases significantly with increasing number of columns or operators.

In a SAP HANA column store table, each column has a main index and a delta index. The main storage is optimized for read operations and the delta storage for write operations. Generally speaking, read operations are performed on both storages and since delta storage is now read-optimized it negatively affects the read operations. The delta storage is used whenever an table gets altered or updated. Therefore it is desired that the delta storage is empty which we can achieve by manually executing a delta merge operation. This operation merges the delta into the main storage. Whenever a table is written to, we execute such a delta merge operation.

There is an internal parse tree depth maximum which is hard-coded and cannot be changed. This limits the size of expressions we are using.

4.2 Experimental Environment

The hardware specifications of the used machine are stated in Table 4.1. All experiments were run on this setting with four Intel Xeon CPU E5-4650 v2, giving 40 cores (4x10) and 80 hardware threads (4x20) in total.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Linux 3.16.0-4-amd64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution</td>
<td>Debian GNU/Linux 8 (jessie)</td>
</tr>
<tr>
<td>CPU Model</td>
<td>Intel Xeon CPU E5-4650 v2 @ 2.40GHz</td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>4</td>
</tr>
</tbody>
</table>

4.2.1 Shared Scan Setup

All experiments refer to a scan on relation $X$ which is an extended TPC-H Orders table of 100 million tuples without any constraints defined. It has an additional unindexed ID attribute $XID$ which allows
us to specify the selectivity of a query, i.e. how many tuples it should process. We control the selectivity with a filter on the XID attribute. The filters are always range values where the lower bound $L$ is fixed and the upper bound computed as:

$$U = L + \frac{\text{selectivity}}{100} \times \text{TABLESIZE}(X)$$

Table 4.2 lists the different settings for the experiments. Relevant factors are the number of concurrent queries and the selectivity of a single query. The reason why we chose selectivity 99% over 100% is because with 100% there can occur undesired optimizations due to certain implementation characteristics. Each experiment is repeated 5 times. The number of queries vary between experiments depending on the runtime of the approaches and are stated explicitly. Other than that, the other settings apply to all shared join experiments.

To avoid confusion or misunderstanding in later sections, we describe our definition of selectivity: A query is very or highly selective, or has a high selectivity (factor), if the result set of the query contains only a few tuples. For example for a relation with 1000 tuples, a very selective query with a very high selectivity of 0.001% selects only a single tuple. The other way around, the result set of a non-selective query with very low selectivity of 99% contains almost all tuples of the relation.

Table 4.2: Setup for Shared Scan Experiments.

<table>
<thead>
<tr>
<th>Concurrency</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectivity [%]</td>
<td>0.001, 0.1, 1, 10, 99</td>
</tr>
<tr>
<td>Relation size</td>
<td>100 million tuples</td>
</tr>
<tr>
<td>Repetitions</td>
<td>5</td>
</tr>
</tbody>
</table>

4.2.2 Shared Join Setup

Each experiment refers to an equi-join between tables $X$ and $Y$ with same characteristics as the $X$ table defined in the previous section. The join keys on both tables are the ORDERKEY attributes which are always unique. 80% of the join key have a match, i.e. a full table join produces a result set of 80 million tuples.

Table 4.3 lists the different settings for the experiments. Relevant factors are again the number of concurrent queries and the selectivity of a query on attributes $XID$ for table $X$ and $YID$ for table $Y$. The number of queries may vary between experiments depending on the runtime of the approaches and are stated explicitly. Other than that, the other settings apply to all shared join experiments.

Table 4.3: Setup for Shared Join Experiments.

<table>
<thead>
<tr>
<th>Concurrency</th>
<th>80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selectivity [%]</td>
<td>0.001, 0.1, 1, 10, 99</td>
</tr>
<tr>
<td>Relation size</td>
<td>100 million tuples</td>
</tr>
<tr>
<td>Repetitions</td>
<td>5</td>
</tr>
</tbody>
</table>

4.3 Query at a Time

The QAT experiments serve as a baseline on how well the sharing performs. This section evaluates the QAT scan and join implementations in SAP HANA. All execution times are presented as relative values of the QAT scan experiment with a single query and selectivity 99%. Which means that this experiment has an execution time of 1 time unit. We call this time unit $\alpha$ and all following execution times are presented relative to it. All execution times are measured relative to these values. We include the presented plots in the later sections for comparison with the sharing approaches.

QAT Scan. For the evaluation, the shared scan experiments are compared to a baseline QAT experiment with queries like Listing 4.1. In order to make sure that we measure only the execution time of the query, without the time of returning the result set, we apply a MAX aggregation function to the final select statement which reduces the result set size to a single tuple.
Listing 4.1: A single scan query in SAP HANA.

```sql
SELECT * FROM X WHERE XID BETWEEN L AND U;
```

In this experiment we measured execution times for numbers of queries 1, 2, 5, 10, 20 and 120 and interpolated to 240 queries. The performance for QAT is depicted in Fig. 4.1. The execution time increases linear with the number of queries as expected.

![Figure 4.1: Relative performance of QAT scan in SAP HANA.](image)

**QAT Join.** Three types of queries are examined for the join experiments. The first type of queries has predicates only on the build relation, the second on both relations, and the third type has predicates on a random relation. Examples of these queries are listed in Listings 4.2 to 4.4. For comparison the experiments are compared to the QAT experiment where every query is one of the just described queries. Again, in order to make sure that we measure only the execution time of the query we apply a MAX aggregation function to the final select statement to reduce the result set size.

Listing 4.2: Query with predicate only on the build relation.

```sql
1 SELECT * FROM X JOIN Y ON X.A = Y.B
2 WHERE XID BETWEEN L_x and U_x;
```

Listing 4.3: Query with predicate on both relations.

```sql
1 SELECT * FROM X JOIN Y ON X.A = Y.B
2 WHERE (XID BETWEEN L_x AND U_x)
3 AND (YID BETWEEN L_y AND U_y);
```

Listing 4.4: Queries with a predicate on either relation (either left or right query).

```sql
1 SELECT * FROM X JOIN Y ON X.A = Y.B
2 WHERE XID BETWEEN L_x AND U_x;
3 WHERE YID BETWEEN L_y AND U_y;
```

In this experiment we measured execution times for numbers of queries 1, 2, 5, 10, 20, 30, 60, 120 and 240. The performance of the QAT join experiments for each type of query is presented in Fig. 4.2. The plots show the execution time dependent on the number of queries for different selectivities on a log-log scale. The left plot represents the first type of queries, the middle plot queries of the second type, and the third type is displayed in the right plot. The behavior of the performance is similar for all three types. The execution has a more or less constant cost per query as expected, so the running time increases linearly with increasing number of queries. A single query alone does not use 100% of the resources efficiently, so adding more queries uses previously unused resources, hence the flat behavior of the curves in the beginning.
4.4 Sharing with Query ID Table

The following evaluation corresponds to the method explained in Section 3.2.

4.4.1 Shared Scan

The shared scan is implemented as stated in Listing 3.1 with minor changes. For this experiment the numbers of queries are 1, 2, 5, 10, 20, 40, 80, 120 and 240. Fig. 4.3 shows the execution time of the QAT scan (left), from Section 4.3, and the shared scan (right) depending on number of queries for different selectivities on a log-log scale.

For the shared scan, the execution time increases exponential contrary to the QAT experiment. The performance for selectivities 0.1%-99% are constantly worse than the ones of QAT. The only observable benefit of this method is for 0.001% where it improves the performance slightly for sharing up to 120 concurrent queries. So for workloads with only a few tuples selected this is a good option.

Even though other selectivities do not improve the scan, it is still possible that together with the benefit of the shared join end performance improves.

Figure 4.3: Relative performance of the shared scan approach with QIDTABLE compared to QAT in SAP HANA.
4.4.2 Shared Join

The shared join is implemented as stated in Listing 3.3 with minor changes. For this experiment the numbers of queries are 1, 2, 5, 10, 20, 60, 120, and 240 for selectivity 0.001%. With decreasing selectivity the number of queries are reduced due to the long runtime of the experiments. Fig. 4.4 shows the execution time for the QAT join (top row), from Section 4.3, and the shared join (bottom row) depending on number of queries for different selectivities on a log-log scale.

After a certain amount of queries the execution time for any selectivity increases drastically. This makes sense, due to the nature of the approach, where tuples are duplicated for every additional query, hence, at some point this overhead will become a bottleneck. For queries with predicates on the build relation or on both relations, this approach outperforms QAT for selectivity 0.001% when processing up to 240 queries. For example, with predicates on the build relation it is almost an order of magnitude faster for 120 queries than QAT, and with predicates on both relation even two orders of magnitude. For the third type of queries, with predicates on either relation, the performance significantly decreases already with two queries. The reason why the curve for selectivity 99% does not align with the others is because the optimizer executes a different execution plan which we cannot influence.

![Graph showing performance comparison between QAT and Shared Join for different predicate selectivities](image)

Figure 4.4: Relative performance of shared join with QIDTABLE compared to QAT in SAP HANA.

4.5 Sharing with Arrays

The following evaluation of the MQJoin implementation corresponds to Section 3.3. SAP HANA does support arrays and some basic manipulation functionality. However, there is no function similar to ARRAYREMOVE, hence, the inline approach cannot remove the zeros which results in a huge overhead for further processing. We also are not able to apply the binary search trick because the CASE expression do not allow to return arrays. Since SAP HANA does not support arrays as return or input types for UDFs, UDFs are not an option either. The shared join is not possible due to the lack of an intersection function.

Regarding the shared scan version with an additional predicates filter, the pre-filtering showed only benefits for SAP HANA. In the other database systems the pre-filter hardly improved the performance, to the contrary, it often resulted in a slightly worse performance. Therefore, we discuss the pre-filtering only for SAP HANA and omit it for the remaining database systems.
Figure 4.5: Relative performance of the shared scan approach with arrays compared to QAT in SAP HANA.

**Shared Scan.** Even though an ARRAY_REMOVE function is not available it is still interesting to see how the shared scan itself performs. In Fig. 4.5a we see the performance of the QAT scan experiments on a log-log scale. The actual inline implementation with arrays is similar to Listing 3.6 without the ARRAY_REMOVE function. When comparing QAT processing to the performances of the inline version, Fig. 4.5b we see that arrays are undesired as the performance in all experiments is worse than QAT processing, and even up to two orders of magnitude worse for low numbers of queries. The reason for that becomes clear when looking at the query execution plan. The optimizer chooses a plan that materializes intermediate results in internal tables. The materialization is very expensive, especially since all tuples are scanned, resulting in the bad performance.

However, when we add a filter with the predicates to the query, the performance drastically improves, as we see in Fig. 4.5c. The filter improves the run time for every selectivity factor except for 99%. This is expected since in this case almost the whole relation is scanned anyway. Even though selectivity factors 0.1% to 10% receive a performance boost, they still are no match for QAT processing. For selectivity factor 0.001%, on the other hand, we now have a configuration that outperforms the QAT processing. The profit increases with increasing number of queries. For a low number of queries the filter achieves a speed up of two to three whereas for 120 queries it is already a order of magnitude faster.

### 4.6 Sharing with Bitsets

The following evaluation corresponds to Section 3.4. As stated in Section 3.4.4, current systems do not have a bitset data type but need to simulate one with integers which are limited in size. The largest
data type bigint (64 bit signed integer) allows us to use 63 bits for storing up to 63 queries. For ease of presentation, only 60 bits are effectively used during the microbenchmarks. The three unused bits have only a minor impact on the performance and can be neglected.

4.6.1 Shared Scan

This section discusses the actual implementations and the behavior of the different shared scan approaches from Section 3.4.1 in SAP HANA. We consider four different approaches: (i) inline, (ii) inline with binary search, (iii) multiple UDFs and (iv) single UDF. The effective implementations hardly differ from the pseudo-codes unless explicitly mentioned. Fig. 4.6 contains the execution times of the approaches depending on the number of queries for different selectivities on a log-log scale. Fig. 4.7 shows the same experiments but with pre-filtering the predicates. The scan baseline experiment from Section 4.3 is depicted again in Fig. 4.6a and Fig. 4.7a. The analysis of the shared scan approaches follows below.

Inline and Binary Search Inline. The inline performance is depicted in Fig. 4.6b. It performs generally worse than QAT. We associate this with column-at-a-time and operator-at-a-time processing, since multiple bitset columns and many operations are involved in the expressions the computation is slow. On the other hand, the binary search implementation, Fig. 4.6c, runs generally faster than, or at least as fast as, the standard inline approach. For selectivity 99% there is a remarkable benefit with at least 20 queries. It performs close to the QAT approach for the other selectivities, about 1.5 times slower.

Figs. 4.7b and 4.7c show the equivalent results of the experiments with pre-filtering for the inline and the inline with binary search approaches respectively. The more selective the queries the more does the filter pay off. Compared to QAT we now see clear improvements for the selectivity 0.001% experiments in both cases. Other than that, for the remaining selectivity factors, even though they perform better than without filtering, they still are slower than QAT. However, the more important observation is that the filter gives in many cases a speed up in the order of magnitude compared to the non-filtered versions.

We hit the internal parse tree depth maximum mentioned in Section 4.1 with the standard inline approach with 480 queries. The same will hold for the binary search algorithm for some number of queries which we do not know yet. So this approach has a hard-coded limitation in SAP HANA.

Multiple UDFs. The performance for multiple UDFs without pre-filtering is depicted in Fig. 4.6d. The runtime is the same for every selectivity and constant for 1 to 60 concurrent queries. This is exactly the number a single UDF can process. As soon as multiple UDFs are needed the execution time increases linear with every additional UDF. The main reason for this behavior is, that UDFs are called per tuple which introduces a significant overhead. This overhead is much higher than the actual computation of the bitsets. This can be observed when comparing Fig. 4.6d with Fig. 4.6e which shows the performance for dummy UDFs that merely return 0. Even though there is no computation involved the dummy UDFs perform almost the same as the ones that compute the bitsets.

The overhead makes UDFs without pre-filtering an uninteresting option when only computing a single bitset. With multiple UDFs, i.e. more than 60 concurrent queries, it runs faster than QAT for selectivity 99% but performs just as good as the inline binary search approach. Otherwise it is slower than the inline binary search approach.

Figs. 4.7d and 4.7e show the equivalent performances of the experiments with pre-filtering for the multiple UDFs and the dummy UDFs approaches respectively. The pay off of the pre-filtering is immense in these cases. Previously the call overhead rendered UDFs impractical but with predicates filtering they suddenly become a valid approach. For selectivity 0.001% it even performs better than the inline binary approach with an speed up factor of two to three for selectivity 0.001%. Compared to QAT processing the 0.001% experiments achieve up to an order of magnitude speed up for 240 queries and the 0.1% experiments perform almost same as for QAT. For selectivity factors 1% to 99% the inline approaches are more desirable as they perform generally better. The overhead of UDFs is still visible in Fig. 4.7e but now is much lower because fewer tuples are processed.

Single UDF. In Listing 4.5 we see the shared scan implementation using a single UDF to compute all
Figure 4.6: Relative performance of different shared scan approaches with bitsets compared to QAT in SAP HANA.
Figure 4.7: Relative performance of different shared scan with predicate filter approaches using bitsets compared to QAT in SAP HANA.
bitsets at once. The DETERMINISTIC keyword is optional.

Listing 4.5: Shared scan implementation using a UDF to compute all bitsets (N

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CREATE FUNCTION BITSETS(IN val INT)</td>
</tr>
<tr>
<td>2</td>
<td>RETURNS bitset0 BIGINT, bitset1 BIGINT [DETERMINISTIC] AS</td>
</tr>
<tr>
<td>3</td>
<td>BEGIN</td>
</tr>
<tr>
<td>4</td>
<td>bitset0 = ...;</td>
</tr>
<tr>
<td>5</td>
<td>bitset1 = ...;</td>
</tr>
<tr>
<td>6</td>
<td>END;</td>
</tr>
<tr>
<td>7</td>
<td>SELECT BITSETS(XID).bitset0, BITSETS(XID).bitset1 FROM X;</td>
</tr>
</tbody>
</table>

The problem with Listing 4.5 without the DETERMINISTIC keyword is that the UDF still gets executed twice instead of only once. It gets executed for each output parameter separately since the computed values are not cached, giving the same performance penalty as before. Even the DETERMINISTIC keyword, which was introduced in SAP HANA 2 to store results for distinct input parameters in cache, does not help here. In this case, for each output parameter SAP HANA produces an evaluation tree even if they are defined in the same UDF. Hence, SAP HANA has to execute multiple evaluation graphs which has a similar performance like multiple UDFs. In that sense, the DETERMINISTIC keyword is best used when accessing the same output parameter with same input values.

As a final conclusion we pick the inline approach with binary search as the most performant implementation for the shared scan as it has the best overall performance of all the tested approaches. We will use it for the shared join in the next section. In the case of pre-filtering with the predicates we will also consider the UDF approach as it performed similar to the inline approaches.

4.6.2 Shared Join

This section discusses the actual implementations and the behavior of the shared join approach from Section 3.4.2 in SAP HANA. For that we examine two versions of the shared join experiments. The first version performs a shared join in isolation without a shared scan. To eliminate the effects of shared scan the bitsets are precomputed and materialized in bitset columns for tables X and Y. The second version computes the shared scans on X and Y using the best shared scan approach from the previous section, in terms of overall performance, which is the inline approach with binary search.

Figs. 4.8 and 4.9 show performances for queries without and with predicates filtering respectively. The join baseline experiment from Section 4.3 is depicted again in the top rows of both figures. Items 8 to 10 below summarize the evaluation of the plots.

Shared Join (w/o Shared Scans). In this shared join version, we generate two calculated bitset columns on each relation which are materialized. However, SAP HANA has a limit on the expression length for calculated columns. Since the length of CASE expressions is restricted, only 50 queries can be stored in one bitset instead of the usual 60 when computing the bitset columns for the shared join without shared scans.

Listing 4.6 shows a shared join in SAP HANA’s SQLScript. It should be noted that intermediate variables are usually inlined during execution and used to improve readability. The join here, however, is not inlined but computed as an actual intermediate table using the NO_INLINE hint because it improves the performance most of the time. It appears that the optimizer obtains additional information that way allowing it to generate a better execution plan. Corresponding performances are depicted in Figs. 4.8 and 4.9 in the second rows for queries without and with predicates filtering respectively.
Listing 4.6: Shared join implementation in SQLScript.

```
DO BEGIN
  i_X = SELECT * FROM X WHERE (bitset0 <> 0 OR bitset1 <> 0);
  i_Y = SELECT * FROM Y WHERE (bitset0 <> 0 OR bitset1 <> 0);
  i_join = SELECT bitset0, bitset1
              FROM :i_X JOIN :i_Y ON :i_X.A = :i_Y.B
              WITH HINTS(NO_INLINE);
  SELECT *
  FROM ( SELECT BITAND(:i_X.bitset0, :i_Y.bitset0) AS final_bitset0,
          BITAND(:i_X.bitset1, :i_Y.bitset1) AS final_bitset1
          FROM :i_join )
  WHERE final_bitset0 <> 0 OR final_bitset1 <> 0;
END;
```

Shared Scan & Join. The shared join with shared scans only differs in lines 2 and 3 which now become:

```
i_X = SELECT * FROM SharedScan(X);
i_Y = SELECT * FROM SharedScan(Y);
```

Corresponding inline performances are depicted in Figs. 4.8 and 4.9 in the third rows for queries without and with predicates filtering respectively. The UDF approach is only practical in combination with pre-filtering the relations before computing the bitsets. Its performance is depicted in Fig. 4.9 in the bottom row.

Figure 4.8: Relative performance of shared join compared to QAT in SAP HANA.
The following list summarizes the general behaviors of the different shared join versions that can be observed in Fig. 4.8:

- **Shared Join (w/o Shared Scans).** For selectivity 99%, the runtime is almost constant and identical for all three types of queries. This makes sense because the worst case scenario, where all tuples from both tables get joined, is the same in all three cases. It is to be expected that with increasing number of queries the curves of all other selectivities converge to the 99% curve, which can be observed. The reason for this behavior is that when having more and more queries at some point all tuple from the tables will be selected which corresponds to the case when selectivity is 99%. The authors of MQJoin [8] observed the same behavior.

- The execution time has no hard ceiling in contrast to the observation in the MQJoin [8] paper. This comes from overheads such as having more and more bitsets with increasing number of queries.

Regarding the experiments with pre-filters, the performances hardly differ from the ones without a filter. This makes sense as the filtering should only affect the shared scans, but in this case there are no shared scans.
ii **Shared Scan & Join (Inline).** The execution times behave roughly as the ones from the shared join without the shared scans experiments plus two times the execution times for a shared scan (one for each relation). After reaching 60 queries the curves start to slope upwards which is attributed to the use of multiple bitsets.

For the first and second types of queries there is small peak for 5 queries. The shared scan experiments show the same behavior and is hence attribute to the shared scans. Another observation is, when comparing to the shared join without shared scans, that the shared scans do have a major impact on the execution time for high selectivities. For low selective queries the join performance dominates but with increasing number of queries the shared scans get more and more important.

As for the experiments with pre-filtering, the performances for queries with predicates on either relation is the same as without using a filter. For queries with predicates on the build relation or on both relations we see a significant improvement. The behavior is similar to the one we observed for shared scans. Additional to very low selective queries, we new also obtain a benefit for high selective queries, and the more selective the query is, the larger is the benefit. This correlates directly to the behavior of the shared scans with pre-filtering. Compared to QAT processing we now get better performances for high selectivities, i.e., 0.001% and 0.1%, up to an order of magnitude.

iii **Shared Scan & Join (UDF).** The UDF approach with pre-filtering has very similar behavior as the inline approach with pre-filtering. For selectivities 0.1% to 99% it performs similar or slightly worse and for 0.001% up to a factor of two to three faster for 240 queries. This behavior coincides with the one for the shared scan experiments.

For all three types of queries, there is a benefit for selectivity 99% when having at least two queries. For high selectivities the shared join shows benefits only for a few configurations.

### 4.7 Conclusion

We have seen that several sharing approaches give a practical performance boost under certain conditions. Using an query ID table gives a considerable benefit when only a few tuples are involved in the shared scan or shared join. The less tuples are scanned or joined, the larger is the benefit. This can be useful when the user know how many tuples are expected from a query or when the involved relations are rather small.

The array method can only be applied for shared scan in the current state of SAP HANA. We could not observe a performance at all for this method. The materialization when using arrays makes the approach unattractive. The binary search algorithm is not possible but might improve the performance significantly as it returns the minimum arrays without any zeros, i.e. it makes the ARRAY\_REMOVE function redundant. Unfortunately we cannot implement a shared join due to missing features.

With bitsets, shared scans can certainly improve performance for high workload queries. The hard bound on the parse tree depth limits the number of concurrent queries though. Unfortunately, UDFs have a constant overhead by just getting called for each tuple. It would be interesting to see how the performance would behave if the DETERMINISTIC keyword would work as we hoped, i.e. the UDF computes all bitsets during a single execution. This could be a significant improvement. The shared join shows better performances than QAT processing for very low selectivities, in order of magnitudes depending on the number of queries. For high selectivities we found that the execution time can sometimes be improved by a factor of up to 2 in some cases.

Pre-filtering shared scans greatly improves the performance not only of the shared scan but also for the shared join that uses it. With predicate filters we also gain performance boost for very highly selective queries compared to QAT processing.

Overall the experiments show that it is very well possible to improve the processing time of queries by sharing common parts between queries using pure SQL in SAP HANA. However, it requires certain prior knowledge of what the queries do, the sizes of tables and the overall workload.

### 5 SQL Server

In this section we discuss the different sharing methods on the Microsoft SQL Server database. SQL Server is a relational database management system that supports row as well column storage. We consider column storage as it performs better for analytical tasks. The experimental setup in this section is
the same as defined in Section 4.2.

The rest of this section is structured as follows: Section 5.1 gives some basic insights into SQL Server. In the second section, Section 5.2, we analyze the query at a time (QAT) baseline experiments. Sections 5.3 and 5.4 evaluate the query ID table and the bitset methods explained in Section 3. Since SQL Server does not support arrays, the corresponding method is not possible to implement. In the last section, Section 5.5, we conclude the evaluation for SQL Server.

5.1 SQL Server Architecture

The database version Microsoft SQL Server 2017 (CTP2.1) - 14.0.600.250 (X64) is used together with the command line tool sqsh-2.1.7 for communication. The following restriction of the SQL Server architecture is relevant for MQE.

SQL Server has a list of parallelism inhibitors that prevent parallel execution either by forcing a serial plan or a serial zone within a parallel plan. One of these inhibitors are UDFs, which force a serial plan.

5.2 Query at a Time

The QAT experiments serve as a baseline on how well the sharing performs. This section evaluates the QAT scan and join implementations in SQL Server. As in Section 4.3, we present all execution times relative to the values of the QAT experiments with a single query and selectivity 99%, for scan ($\alpha_{\text{scan}}$) and join ($\alpha_{\text{join}}$) respectively. We include the presented plots in the later sections for comparison with the sharing approaches. The scan and join experiments are executed with the same setup as described in Section 4.3.

**QAT Scan.** The performance for QAT is depicted in Fig. 5.1, which shows the execution time dependent on the number of queries for different selectivities on a log-log scale. The execution time increases linear with the number of queries as expected.

![Figure 5.1: Relative performance of QAT scan in SQL Server.](image)

**QAT Join.** The performances of the QAT join experiments for each type of query are presented in Fig. 5.2. The plots show the execution time dependent on the number of queries for different selectivities on a log-log scale. The execution has a more or less constant cost per query as expected, so the running time increases linearly with increasing number of queries.

5.3 Sharing with Query ID Table

The following evaluation corresponds to the method explained in Section 3.2.
5.3.1 Shared Scan

The shared scan is implemented as stated in Listing 3.1 with minor changes. For this experiment the numbers of queries are 1, 2, 5, 10, 20, 40, 80, 160, and 320. Fig. 5.3 shows the execution time of the QAT scan (left), from Section 5.2 and the shared scan (right) depending on number of queries for different selectivities on a log-log scale.

The runtime increases exponential with increasing numbers of queries. The only beneficial configurations for this approach is with a really high selectivity, i.e. only for 0.001%, and only until with up to 80 queries. This is not a surprising result due to the characteristic behavior of the method as stated in Section 3.2.

Figure 5.3: Relative performance of the shared scan approach with QIDTABLE compared to QAT in SQL Server.

5.3.2 Shared Join

The shared join is implemented as stated in Listing 3.3 with minor changes. The only difference is that the intermediate results are defined in a SQL WITH clause instead. For these experiments the numbers of queries depend on the selectivity which affects the runtime extremely. For low selectivities we execute only a few queries whereas for high selectivities more queries are tested, due to the run time of the experiments. The different numbers of queries are 1, 2, 5, 10, 20, 40, 80, 160, and 320. Fig. 5.4 shows the execution time for the QAT join (top row), from Section 5.2 and the shared join (bottom row)
depending on number of queries for different selectivities on a log-log scale.

![Graph showing relative performance of shared join with QIDTABLE compared to QAT in SQL Server.](image)

Figure 5.4: Relative performance of shared join with QIDTABLE compared to QAT in SQL Server.

The runtimes increase exponentially as expected. The approach shows high runtimes for low selectivities. With 10% and 99% tuples selected, the runtime is already unattractive with only one or two queries. In the left plot we can see a slight benefit for selectivity 0.001% between 2 and 160 queries. In some cases, e.g. for 20 queries, it halves the runtime. The middle plot shows benefits in more cases compared to QAT processing. For selectivities 1%, 0.1% and 0.001% the performance improves for up to 10, 40 and 320 concurrent respectively. After those points the performance drops drastically. For queries with predicates on either relation the approach is unpractical. The overhead of duplication generates just too much workload for the join with low selectivities.

Overall this approach can turn out to be useful when the selectivity of the queries is known. For small numbers of selected tuples it runs up to three times faster than when executing the queries one by one.

### 5.4 Sharing with Bitsets

The following evaluation corresponds to Section 3.4. The largest data type `bigint` (64 bit signed integer) allows us to use 63 bits for storing up to 63 queries. For ease of presentation, only 60 bits are effectively used during the microbenchmarks. The three unused bits have only a minor impact on the performance and can be neglected.

#### 5.4.1 Shared Scan

This section discusses the behavior of the inline, with and without binary search, shared scan approaches from Section 3.4.1 in MySQL. The effective implementations hardly differ from the pseudo-codes. We discuss the performances for 1, 2, 4, 8, 15, 30, 60, 120 and 240 concurrent queries. Fig. 5.5 contains the execution times of the approaches depending on the number of queries for different selectivities on a log-log scale. The scan baseline experiment from Section 5.2 is depicted again in Fig. 5.5a. The analysis of the shared scan approaches follows below.

The standard inline performance is depicted in Fig. 5.5b. For the first 60 queries, it performs similar to the QAT processing with a small overhead from the `CASE` expressions. For more than 60 queries the runtime rises by roughly a factor of two.
Figure 5.5: Relative performance of the different shared scan approaches compared to QAT execution in SQL Server.

The binary search implementation, Fig. 5.5c, on the other hand, runs faster than the standard inline approach as well as the QAT processing. Especially for more than two queries the performance benefits significantly from this approach. E.g. for 120 concurrent it performs an order of magnitude faster than QAT processing. We see the same characteristic spike from 60 to 120 queries which roughly doubles the runtime, same as the increase in numbers of bitsets.

As a final conclusion we pick the inline approach with binary search for the shared scan as it has the better overall performance. We will use it for the shared join in the next section.

### 5.4.2 Shared Join

This section discusses the actual implementations and the behavior of the shared join approach from Section 3.4.2 in SQL Server. The implementation is similar to the pseudo-code with minor syntax changes and the added `MAX` aggregation. We examine the same two versions of shared join experiments as described in Section 4.6.2 and discuss the performances for 1, 2, 5, 10, 30, 60, 120 and 240 concurrent queries. The join baseline experiment from Section 5.2 is depicted again in Fig. 5.6 in the top row.

**Shared Join (w/o Shared Scans).** To measure the performance of a shared join without the effects of shared scans, we add new bitset columns on each relation which are materialized. So instead of computing the bitsets on the fly, we use these materialized columns. The actual implementation differs from the pseudo-code in that the intermediate results \(_i \cdot X\) and \(_i \cdot Y\) are defined in a SQL `WITH` clause instead.
Figure 5.6: Relative performance of shared join, with and without shared scans, compared to QAT processing in SQL Server.

**Shared Scan & Join.** The shared join with shared scans now compute the actual shared scan on the fly. Also the computed bitsets need a cast to bigint for the bitwise operation, SQL Server uses decimal as default type for values larger than the maximum integer value. Corresponding performance plots are found in Fig. 5.6 in the bottom row.

The following list summarizes the general behaviors of the different shared join versions that can be observed in Fig. 5.6:

1. **Shared Join (w/o Shared Scans).** For selectivity 99%, the runtime behaves identical for all three types of queries. This makes sense because the worst case scenario of a full table join is the same in all three cases. It is to be expected that with increasing number of queries the curves of all other selectivities converge to the 99% curve, which can be observed. The reason for this behavior is that when having more and more queries at some point all tuple from the tables will be selected which corresponds to the case when selectivity is 99%. This is also the case for all experiments with at least two queries with predicated on either relation. Therefore, the performance for those experiments are the same as shown in the most right plot. The authors of MQJoin [8] observed the same behavior. As for SAP HANA the execution time has no hard ceiling due to multiple bitsets with increasing number of queries. Until 60 queries the curves proceed constant. After reaching 60 queries the curves start to slope upwards which is attributed to the use of multiple bitsets.

2. **Shared Scan & Join.** The execution times behave roughly as the ones from the shared join without the shared scans experiments plus the execution times for the shared scans.

Another observation is, when comparing to the shared join without shared scans, that the shared scans do have a major impact on the execution time for highly selective queries.
For all three types of queries, there is a visible benefit for each selectivity with at least five concurrent queries.

5.5 Conclusion

The query ID table method shows only benefits for very low workload. The best use case would be for queries operating on tables of small size.

We have seen that MQE gives a considerable performance boost of up to two orders of magnitude for the bitset method. The behavior of the shared scan, which improves the performance by orders of magnitudes for increasing numbers of queries, is reflected in the join as well. Interestingly, this applies to all kind of workloads. With at least 10 concurrent queries we obtain a performance speed up for all selectivities and for all three types of queries.

6 MonetDB

The following section discusses the different sharing methods on the MonetDB database. MonetDB is a read optimized, columnar database management system. The experimental setup in this section is the same as defined in Section 4.2.

The rest of this section is structured as follows: Section 6.1 gives some basic insights into internal structure of MonetDB. In the second section, Section 6.2, we analyze the query at a time (QAT) baseline experiments. Sections 6.3 and 6.4 evaluate the query ID table and the bitset methods explained in Section 3. Since MonetDB does not support arrays, the corresponding method is not possible to implement. In the last section, Section 6.5, we conclude the evaluation for MonetDB.

6.1 MonetDB Architecture

The database version MonetDB v11.25.23 (Dec2016-SP5) is used together with mclient for client side communication. The following characteristics of the MonetDB architecture are relevant for MQE.

MonetDB stores each column of a relation in a single memory heap that represents a c-array where the first heap entry corresponds to the first tuple, i.e. the position in the heap implies the row ID of the tuple. It uses arrays as they are the most natural way for CPUs to understand and to write code. But they are also useful for fast look-ups and for optimizers.

MonetDB does operator-at-a-time processing. Each operator consumes and produces entire columns, which are then processed by other operators. Each intermediate result is fully materialized in memory, though latest MonetDB versions use lazy materialization.

MonetDB either sorts data (binary search benefits from this) or uses hash-indexes or column imprints [12], a cache conscious secondary index, to speed up range queries. Imprints is a data skipping index that skips cachelines where the data definitely not resides, hence, it produces false-positives which can introduce a bit of an overhead.

6.2 Query at a Time

The QAT experiments serve as a baseline on how well the sharing performs. This section evaluates the QAT scan and join implementations in MonetDB. Same as in Section 4.3 we present all execution times relative to the values of the QAT experiments with a single query and selectivity 99%, for scan ($\alpha_{\text{scan}}$) and join ($\alpha_{\text{join}}$) respectively. We include the presented plots in the later sections for comparison with the sharing approaches. The scan and join experiments are executed with the same setup as described in Section 4.3.

QAT Scan. The performance for QAT is depicted in Fig. 6.1 which shows the execution time dependent on the number of queries for different selectivities on a log-log scale. The execution time increases linear with the number of queries as expected. Surprisingly, the 10% selectivity microbenchmark perform best even though the same plan is executed. Without further investigation we cannot make a clear point on this behavior.
Another interesting observation is that the performances between the selectivities hardly differ. We attribute this to the storage model of using arrays, and the effect of imprints that are designed to speed up scans.

![Figure 6.1: Relative performance of QAT scan in MonetDB.](image)

**QAT Join.** The performances of the QAT join experiments for each type of query are presented in Fig. 6.2. The plots show the execution time dependent on the number of queries for different selectivities on a log-log scale. The left plot represents the first type of queries, the middle plot queries of the second type, and the third type is displayed in the right plot. The execution has a more or less constant cost per query as expected, so the running time increases linearly with increasing number of queries. Apparently the performance is quite similar for all three types of queries. We expect to see a better performance for queries with predicates on both relations as the join there processes less tuples. However, this is only observable for selectivity 10%. When comparing the curves with those of the QAT scan we realize that they are pretty similar in run time. This means that the major performance factor of the join comes from the scans, which also explains the previously stated behavior.

![Figure 6.2: Relative performance of QAT join in MonetDB.](image)

### 6.3 Sharing with Query ID Table

The following evaluation corresponds to the method explained in Section 3.2.

#### 6.3.1 Shared Scan

The shared scan is implemented as stated in Listing 3.1 with minor changes. For this experiment the numbers of queries are 1, 2, 5, 10, 20, 40, 80 and 160. Fig. 6.3 shows the execution time of the QAT.
The overall performance of the shared scan for all selectivities is worse than the QAT processing already starting for two queries. After a certain amount of concurrent queries the execution times start increasing, e.g. at 20 queries for selectivity 99% or 80 queries for selectivity 10%. This relates to the habitual behavior of the method as stated in Section 3.2.

6.3.2 Shared Join

The shared join is implemented as stated in Listing 3.3 with minor changes. For this experiment the numbers of queries are 1, 2, 5 and 10 as it already shows the trend of the performance. Fig. 4.4 shows the execution time for the QAT join (top row), from Section 4.3, and the shared join (bottom row) depending on number of queries for different selectivities on a log-log scale. The execution time of the shared join is in almost all cases orders of magnitudes larger than for QAT processing, rendering this method unpractical for MonetDB.

6.4 Sharing with Bitsets

The following evaluation corresponds to Section 3.4. The largest data type bigint (64 bit signed integer) allows us to use 63 bits for storing up to 63 queries. For ease of presentation, only 60 bits are effectively used during the microbenchmarks. The three unused bits have only a minor impact on the performance and can be neglected.

6.4.1 Shared Scan

This section discusses the behavior of different shared scan approaches from Section 3.4.1 in MonetDB. Since in MonetDB UDFs are not parallelized we analyze just the two inline approaches. The effective implementations hardly differ from the pseudo-codes. Fig. 6.5 contains the execution times of the approaches depending on the number of queries for different selectivities on a log-log scale. The scan baseline experiment from Section 4.3 is depicted again in Fig. 6.5a. The analysis of the shared scan approaches follows below.

The standard inline performance is depicted in Fig. 6.5b. It performs generally worse than QAT. We associate this with operator-at-a-time processing, since many operations are involved in the expressions the computation is slow and gets worse for increasing number of queries.
Figure 6.4: Relative performance of shared join with QIDTABLE compared to QAT in MonetDB.

The binary search implementation, Fig. 4.6c, on the other hand, runs slightly faster than, or at least as fast as, the standard inline approach. Even though the binary search algorithm should give an increasing benefit with increasing number of queries, for selectivity 99% it performs almost the same as without. The best performance is achieved for high selectivities and up to 10 queries, which is close to the QAT processing, but the performance gets worse with more queries.

The inline with binary search is the most efficient approach. Although it gives not much of an improvement, it is still possible that, together with the benefits of a shared join, the end performance might improve.

6.4.2 Shared Join

This section discusses the actual implementations and the behavior of the shared join approach from Section 6.4.2 in MonetDB. We examine the same two versions of the shared join experiments as described in Section 4.6.2. The join baseline experiment from Section 6.2 is depicted again in Fig. 6.6 in the top row. Items ii and iii below summarize the evaluation of the plots.

Shared Join (w/o Shared Scans). To measure the performance of a shared join without the effects of shared scans, we add bitset columns to each relation and update the bitsets for each tuple with the corresponding bitset value. So instead of computing the bitsets on the fly, we use these materialized columns. The actual implementation differs from the pseudo-code in that the intermediate results \( i_X \) and \( i_Y \) are defined in a SQL \( WITH \) clause instead.

Shared Scan & Join. The version differs from the previous in defining \( i_X \) and \( i_Y \), which now compute the actual shared scans. Corresponding performance plots are found in Fig. 6.6 in the bottom row.

The following list summarizes the general behaviors of the two shared join versions that can be observed in Fig. 6.6

i Shared Join (w/o Shared Scans). It is to be expected that with increasing number of queries the curves of all other selectivities converge to the 99% curve, which can be observed. The reason for this behavior is that when having more and more queries at some point all tuple from the tables will be selected which corresponds to the case when all tuples are processed. The authors of MQJoin observed the same behavior.
Figure 6.5: Relative performances of the different shared scan approaches compared to QAT execution in MonetDB.

The execution time has no hard ceiling in contrast to the observation in the MQJoin paper. This comes from overheads such as having more bitsets with increasing number of queries.

ii Shared Scan & Join. The shared scans add an large performance penalty such that the performance is now up to an order of magnitude worse than the QAT processing. For all three types of queries, the shared join gives no benefit at all.

6.5 Conclusion

Our shared scan implementation did not result in an improve in MonetDB, whether with the query ID table method or with bitsets. For the bitsets, we attribute this to the operator-at-a-time processing as our method contains many operators. For a more thorough understanding a deeper investigation is needed with a deeper knowledge of MonetDB’s architecture.

7 MySQL

In this section we discuss the different sharing methods on the MySQL database. MySQL is a database management system designed for OLTP workloads. We chose MySQL to demonstrate the behavior of our methods in a single-threaded, transactional system. The experimental environment follows the same layout as in Section 4.2. However, since MySQL does not provide intra-query parallelism some of the settings change. The first difference comes directly from the fact that there is no intra-query parallelism,
Figure 6.6: Relative performance of shared join, with and without shared scans, compared to QAT in MonetDB.

Therefore, the concurrency is limited to a single thread per query. So in the following experiments there is no parallelization involved. The other two changes, which are indirectly related to this as well, are that we reduced the relation sizes to 1 million tuples and add a primary key to the join attributes. Otherwise the microbenchmarks would consume too much time.

The rest of this section is structured as follows: Section 7.1 gives some basic insights into the internal structure of MySQL. In the second section, Section 7.2, we analyze the query at a time (QAT) baseline experiments. Sections 7.3 and 7.4 evaluate the query ID table and the bitset methods explained in Section 3. Since MySQL does not support arrays, the corresponding method is not possible to implement. In the last section, Section 7.5, we conclude the evaluation for MySQL.

7.1 MySQL Architecture

The database version 5.7.18 MySQL Community Server (GPL) is used together with the command line tool mysql for communication. The following characteristics of the MySQL architecture are relevant for the experiment setup.

MySQL uses connection manager threads, which associate each client connection with a thread. As we use only a single client, there is no parallelization with our setup and implementations.

MySQL has a pluggable storage engine architecture, which allows us to select specialized storage engines for particular applications. We use the general-purpose storage engine InnoDB, which stores relational tables in a row storage. MySQL uses InnoDB by default.
7.2 Query at a Time

The QAT experiments serve as a baseline on how well the sharing performs. This section evaluates the QAT scan and join implementations in MySQL. Same as in Section 4.3, we present all execution times relative to the values of the QAT experiments with a single query and selectivity 99%, for scans ($\alpha_{\text{scan}}$) and joins ($\alpha_{\text{join}}$) respectively. We include the presented plots in the later sections for comparison with the sharing approaches. The scan and join experiments are executed with the same setup as described in Section 4.3.

**QAT Scan.** The performance for QAT is depicted in Fig. 7.1 which shows the execution time dependent on the number of queries for different selectivities on a log-log scale. The execution time increases linear with the number of queries as expected.

![Figure 7.1: Relative performance of QAT scan in MySQL.](image)

**QAT Join.** In this experiment we measured execution times for numbers of queries 1, 2, 5, 10, 20, 40, and interpolated for larger numbers of queries. Fig. 7.2 shows the performances of the QAT join experiments for each type of query. The plots show the execution time dependent on the number of queries for different selectivities on a log-log scale. The execution has a more or less constant cost per query as expected, so the running time increases linearly with increasing number of queries.

![Figure 7.2: Relative performance of QAT join in MySQL.](image)

7.3 Sharing with Query ID Table

The following evaluation corresponds to the method explained in Section 3.2.
7.3.1 Shared Scan

The shared scan is implemented as stated in Listing 5.1 with minor changes. For this experiment the numbers of queries are 1, 2, 5, 10, 20, 40, 80, and additional data points are measure for experiments with low runtime, e.g. for selectivity 0.001% up to 320 queries are tested. Fig. 5.3 shows the execution time of the QAT scan (left), from Section 6.2, and the shared scan (right) depending on number of queries for different selectivities on a log-log scale.

The execution times of the shared scan have an exponential slope. This approach shows an improvement for all selectivities dependent on the number of queries. The less tuples get selected the larger is the benefit. The performance for selectivity 99% is slightly improved for up to 20 concurrent queries, but for selectivity 0.001% we still get an improved performance of factor 3 for 320 concurrent queries. We assume that at some point the execution time of the shared scan will exceed that of the QAT processing. Unfortunately, we cannot explain why the performance for selectivity 99% becomes better than for 10% with 20 queries. However, this might be related to the fact that we have a system optimized for OLTP workload.

![Graph showing execution time vs number of queries for different selectivities](image_url)

Figure 7.3: Relative performance of the shared scan approach with QIDTABLE compared to QAT in MySQL.

7.3.2 Shared Join

The shared join is implemented as stated in Listing 3.3 with minor changes. The only difference is that the intermediate results are defined in a SQL \textit{WITH} clause instead. For this experiment the numbers of queries are 1, 2, 5, 10, 20, 40, 80, and 160. Additional data points are measured for experiments with queries of the first two types. Fig. 7.4 shows the execution time for the QAT join (top row), from Section 7.2, and the shared join (bottom row) depending on number of queries for different selectivities on a log-log scale.

Contrary to the QAT processing, the execution time increases exponentially with increasing numbers of queries. This is not surprising considering the nature of the method, i.e. duplication of tuples per query. Impressive is the speed up for high selectivities though.

The plots show benefits for queries with predicates on the build relation or on both relations up to a certain number of concurrent queries. Looking at the selectivity 0.001% curve there is a performance speed up of up to factor 10 for up to 120 queries. With 40 queries there is even a speed up of factor 10 observable. From the trend of the curves, it becomes clear that at some point, when the workload and the overhead of duplication are too high, they will exceed the QAT performances. Depending on the selectivity, this limit is reached sooner or later, e.g., for selectivity 1%, the limit is already reached with 40 queries, whereas for selectivity 0.001% there is still a significant benefit with 120 queries. Interestingly the plots do not show a benefit for selectivity 10% but again for 99%.
Figure 7.4: Relative performance of shared join with QIDTABLE compared to QAT in MySQL.

For queries with predicates on a random relation there is a considerable performance boost for selectivity 99%. All other selectivities do not benefit from this approach in this case.

7.4 Sharing with Bitsets

The following evaluation corresponds to Section 3.4. The largest data type bigint (64 bit signed integer) allows us to use 63 bits for storing up to 63 queries. For ease of presentation, only 60 bits are effectively used during the microbenchmarks. The three unused bits have only a minor impact on the performance and can be neglected.

7.4.1 Shared Scan

In this section, we discuss the behavior of the inline, with and without binary search, shared scan approaches from Section 3.4.1 in MySQL. The effective implementations hardly differ from the pseudocodes. We discuss the performances for 1, 2, 5, 10, 30, 60, 120, and 240 concurrent queries. Fig. 7.5 contains the execution times of the approaches depending on the number of queries for different selectivities on a log-log scale. The scan baseline experiment from Section 7.2 is depicted again in Fig. 7.5a. The analysis of the shared scan approaches follows below.

The standard inline performance is depicted in Fig. 6.5b. It performs constantly better than QAT for any number of queries and selectivity. A single query does not fully utilize the resources yet, hence the flat behavior at the beginning. The binary search implementation improves the runtime by another significant amount. For the first 60 queries the runtime is almost constant and only increases with additional bitset columns. With 30 queries and onwards the execution time is already an order of magnitude faster than with QAT processing, for 240 queries even two orders of magnitude.

7.4.2 Shared Join

In this section, we discuss the actual implementations and the behavior of the shared join approach from Section 3.4.2 in MySQL. The implementation is similar to the pseudo-code with minor changes. We examine the same two versions of shared join experiments as described in Section 4.6.2 and discuss the performances for 1, 2, 5, 10, 30, 60, 120 and 240 concurrent queries. The join baseline experiment from Section 7.2 is depicted again in Fig. 7.6 in the top row.
Figure 7.5: Relative performances of the different shared scan approaches compared to QAT execution in MySQL.

The following list summarizes the general behavior of the two shared join versions that can be observed in Fig. 6.6:

i **Shared Join (w/o Shared Scans).** For selectivity 99%, the runtime is almost constant and identical for all three types of queries. This makes sense because the worst case scenario of a full table join is the same in all three cases. It is to be expected that with increasing number of queries the curves of all other selectivities converge to the 99% curve, which can be observed. The reason for this behavior is that when having many queries at some point all tuples will be selected which corresponds to the case when selectivity is 99%. The authors of MQJoin observed the same behavior.

The execution time has no hard ceiling in contrast to the observation in the MQJoin paper. This comes from overheads such as having multiple bitsets with increasing number of queries. In the case where queries have predicates only on the build relation, there are some fluctuations visible, e.g., for selectivity 1% with 5 queries. The optimizer choses a different execution plan in those cases. The join order, whether X or Y should be the build relation, is reverted compared to the other cases. For these situations, MySQL has an STRAIGHT_JOIN operator that allows to force the left join table to be always read before the right table. Unfortunately, this did not work in our case and is an already known bug.

ii **Shared Scan & Join.** The execution times behave roughly as the ones from the shared join without the shared scans experiments plus the execution times for the shared scans. After reaching 60 queries
the curves start to slope upwards which is attributed to the use of multiple bitsets. With increasing workload the curves converge to the selectivity 99% which is the worst case scenario.

The cost per query of the shared join is much less than for QAT processing, giving a significant performance boost. The benefit compared to QAT processing is immense and for large numbers of queries the shared join is one to two orders of magnitudes faster than QAT processing.

7.5 Conclusion

We have seen that our methods also have considerable impact on non multi-threaded, OLTP systems which are not ideally designed for these methods. The query ID table method gives limited benefits depending on the workload. Due to its natural behavior of the cross product the runtime increases exponential. For a practical use one must consider table sizes and workload of the queries to be able to use it efficiently.

The bitset method, on the other hand, has an significant effect on the performance for the scan and likewise for the join. The behavior of the shared scan, which improves the performance by orders of magnitudes for increasing numbers of queries, is reflected in the join as well. Interestingly, this applies to all kind of workloads. With at least 10 concurrent queries we obtain a performance speed up for all selectivities and for all three types of queries.

Overall the experiments show that it is very well possible to improve the processing time of queries by sharing common parts between queries using pure SQL in MySQL. Using bitsets should be certainly considered when dealing with simple scans and joins.
8 Conclusion and Future Work

From the presented experiments and evaluations, we can conclude that MQE is an effective way to improve existing applications without modifying any existing DBMS code. Depending on internal implementation details of the underlying DBMS, we can achieve performance improvements of orders of magnitude. The order of improvement depends on several parameters. For example, column-at-a-time or operator-at-a-time processing appear to negatively influence the performance benefits. Pre-filtering the shared scans gave only a speed up in SAP HANA, so it seems that now every DBMS finds the opportunities provided to find better query plans. Array are database-specific structures and not always supported in existing DBMS, or at least the necessary functionality for our method is missing. In most database systems, UDFs are not parallelized. Solely SAP HANA used parallelism with UDFs but without pre-filtering the relations the low-overhead call to UDFs rendered this approach unattractive.

The bitset method has the most practical impact as it usually gives an improvement independent of the workload, contrary to the query ID table method. We have shown that MQE can be a viable option with today's systems. If DBMS were to be modified, a bitset data type would be very desirable and should improve the performance even more. We expect it to eliminate the negative effect with column-at-a-time processing because only a single bitset column needs to be computed then. The join would profit from a bitset data type as well, as it simplifies filter expressions and bitwise operations. Additional bit manipulation functionality can also be very useful for the shared aggregation step.

Several topics can be researched in the future. As we did not define any constraints on the relations, we expect that indexes on predicate or join attributes enhance the run times even further, e.g., when applying a pre-filter to the shared scan. We have seen that SAP HANA shows great performance boosts when applying a predicate filter to the scans, which was not the case in the other DBMS. This is also an interesting observation that needs more investigation why this happens. Since in this work we rewrote the SQL queries manually, it would certainly be useful to have a generator that combines arbitrary queries which have common subexpression.

The most interesting research topic would be, how the presented methods behave for more complex queries such as the TPC-H Benchmark. To achieve this, it is possible that a hybrid of the method is needed where some parts of the query need to be shared by a query ID table and other parts with bitsets. For this one must take into account the different dis-/advantages of each method. We presented some approaches how TPC-H Q6 and Q20 can be rewritten and showed that it is not a trivial task. We did some small experimenting but did not have enough time to properly benchmark and evaluate it. Instead we will just describe some things we observed.

A major concern with more complex is the fact that usually a query has predicates on multiple attributes. This is problematic for the binary search algorithm to work, so this can be further researched how to improve or adapt the binary search algorithm to multiple attributes. We also figured that in such a case the order of the evaluation of predicates makes a significant difference. We assume that it depends on the selectivity of the predicates, but this needs further investigation.
A SQL Rewriting Pitfalls

A.1 IN Condition

In this example we explain how the SQL IN condition, that is used in TPC-H Q20 in Section 3.5.2, can be rewritten as an implicit join. Listing A.1 shows the original query with a simple IN condition. In a first step, we replace the IN statement by an implicit join, i.e. we add the relation of the inner subquery to the out query and add the filter \( ps\_suppkey = p\_suppkey \) instead. The second step is to extract the predicate from the condition. For that we simply take the predicate from the inner subquery and append it as a filter to the WHERE clause of the outer query. The final rewritten query look like Listing A.2. In Listing A.3 we illustrate the same query after applying the query ID table method for two queries. The major adjustment here is the GROUP BY at the end. This is needed to separately compute the inner subquery (of the original query) for each query individually.

Listing A.1: Original query.

```
select
  ps_suppkey
from
  partsupp
where
  ps_partkey in (
    select p_partkey
    from part
    where p_name like '[Q_COLOR]%'
  );
```

Listing A.2: IN condition rewritten as implicit join.

```
select
  ps_suppkey
from
  partsupp , part
where
  p_name like '[Q_COLOR]%'
  and ps_partkey = p_partkey;
```

Listing A.3: Shared IN condition rewritten as implicit join using QIDTABLE method.

```
select
  ps_suppkey , qid
from
  partsupp , part , qidtable
where
  qid <= 2
  and (
    (p_name like '[Q1_COLOR]%' and qid = 1) or
    (p_name like '[Q2_COLOR]%' and qid = 2)
  )
  and ps_partkey = p_partkey
group by
  ps_suppkey , qid;
```
Acknowledgements

I want to thank my thesis supervisors Dr. Ingo Müller, Dr. Darko Makreshanski and Prof. Dr. Gustavo Alonso for giving me the opportunity to finish my graduation with an exciting and educational project. Special thanks go to Dr. Ingo MĂżller for supporting me in difficult situations with the right advices and always having the door to his office open for me. My gratitude also goes to Dr. Darko Makreshanski, Renato Marroquin, Lefteris Sidirourgos, Norman May and Robin Rehrmann for sharing their insights and experience with me and/or test reading my work. Another thanks goes to Lars Breddemann from Stackoverflow for fast responses and validating some of the issues I experienced during the project.

I also want to give my thanks to Samuel Hurni, Samuel Schmid, Maximilian Engensperger, Nicolas Forster and Michael Giger for entertaining me with enjoyable activities and chats.

Finally, my gratitude goes to my family for supporting me in many ways. To my father, for encouraging me, helping out with organizational stuff and giving useful advices (even if I did not always follow them). To my mother that always cheers me up and tries to help me wherever she can. To my sisters, for distracting me when I have been working too long which was a welcomed change. To my little niece which is a bundle of joy and made me smile even in the most unpleasant situations.
References


