University of Magdeburg

Faculty of Computer Science

Master's Thesis

# A Hybrid Multi-Query Optimization Technique Combining Sub-Expression and Materialized View Reuse

Author:

Vasudev Raghavendra Bidarkar

4 May 2022

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Balasubramanian Gurumurthy
Department of Computer Science

Prof. Dr.-Ing. Thilo Pionteck
Department of Electrical Engineering and Information Technology

**Raghavendra Bidarkar, Vasudev:**
*A Hybrid Multi-Query Optimization Technique Combining Sub-Expression and Materialized View Reuse*
Master's Thesis, University of Magdeburg, 2022.

# Abstract

Query execution in traditional database systems is sequential and isolated. Such query execution does not reuse results generated for aiding the execution of subsequent queries. Though this type of execution might not present potential gains in performance for OLTP (Online Transaction Processing) workloads, re-using results is beneficial for OLAP (Online Analytical Processing) workloads as queries share some similarity. Multi-Query Execution (MQO) achieves this by generating shared execution strategies for multiple queries. Similarities between queries are used by the Multi-Query optimizer to create an efficient shared query execution plan that may either be batched or use cached information to optimize execution. Unfortunately, both batched and realtime multi-query processing techniques have certain inherent limitations. In this work, we merge these two techniques to form the basis for our *hybrid MQO technique*. By generating shared query plans and using materialized views for deriving query results, we show that our hybrid MQO technique is suitable for multiple query execution in derivable workloads. We also study the role that different operators play in the performance of our hybrid system. Finally, we evaluate the effect that varying cache sizes and derivabilities has on our hybrid MQO system, showing gains in performance over sequential execution for larger caches and higher derivabilities. Our results show that with a generously sized cache, workloads that contain similar queries benefit from using our hybrid method, with an observed speed-up of 2x over sequential execution in the best case.

# Acknowledgements

Firstly, I thank Prof. Gunter Saake for allowing me to write this thesis under his supervision. I would like to express my heartfelt gratitude to my supervisor, Balasubramanian Gurumurthy, whose constructive guidance and helpful suggestions have been invaluable in developing this thesis. I thank Bala for his patience with my many questions and for his crucial insight since the beginning of this work. I am grateful also to David Broneske and Gabriel Campero Durand for their advice, which has been pivotal in helping to better structure and present the concepts in this work.

None of this would have been possible without the support of my parents. I thank them for their constant push for me to try harder. Special thanks also to my friends for their hilarious antics, great food, and unending means of entertainment.

Countless people have contributed in shaping this thesis, many of whom I do not know, but whose tools, books, software, papers, and resources I have relied upon over the duration of this work. I wish to thank everyone who has helped and shaped this work, directly or otherwise.

# Contents

# List of Figures

# List of Algorithms

# List of Abbreviations

**CNF** Conjunctive Normal Form.

**IR** Intermediate Results.

**IRR** Intermediate Results Reuse.

**LRU** Least Recently Used.

**MQO** Multi-Query Optimization.

**MVR** Materialized View Reuse.

**SSE** Shared Sub-Expressions.

# 1 Introduction

In many use-cases, users submit multiple queries to a DBMS in a short time. Although in most situations these queries do not exhibit any particular characteristics, for some systems — such as for Online Analytical Processing (OLAP) — the queries share some degree of similarity, due to their analytical nature [GAK12]. Queries might be similar in that some might read data from the same table, or join over the same predicates.

Traditional database systems do not consider inter-query similarities when generating query execution plans [NBV13], which places them at a disadvantage for processing workloads with similarities. Multi-Query Optimization (MQO) resolves this issue by generating execution strategies that are shared among a given set of queries.

Multiple queries can be processed by a multi-query optimizer in one of two ways. The first approach involves batching queries, mainly based on time, and then computing a shared execution plan to execute all the queries in the batch efficiently. On the other hand, in the second approach queries are processed in a realtime fashion, with each query being optimized individually.

Both MQO techniques have their merits and drawbacks. Batched processing generally does not involve persisting information, thereby merely shifting the problem of isolated execution from query-level to batch-level. Conversely, realtime techniques store information but lack the narrower focus on optimization provided by batched MQO.

In this thesis, we propose a hybrid MQO technique that unifies the principles behind both batched and realtime processing to efficiently and quickly execute queries in shared contexts.

### Multi-Query Optimization

Traditionally, queries are executed in a linear fashion [NBV13], undergoing a series of steps after their submission, culminating in the result being made available to the user. Initially, the query is parsed and converted to an easily operable format, usually a tree or a graph. A logical plan is then created, which is converted into a physical plan in two stages. Firstly, optimization rules are applied to obtain equivalent orderings of the logical plan. Secondly, cost estimates, derived from the query information and the

underlying data, are used to assign costs to plans. Finally, the plan with the lowest overall cost is selected and executed, providing the result to the user.

This process is carried out for every single query separately, which is not ideal for situations where queries supplied to the system are similar. The optimization phase, in particular, can benefit from further improvements to consider not just the present query but also similar ones in its surrounding context.

Multi-Query Optimization (MQO) deals with the problem of executing multiple queries efficiently. The goal of MQO, unlike traditional execution, is to find an efficient approach to executing a set of queries. Conventional MQO techniques can be broadly classified into two categories: batched and realtime. In batched execution, queries are submitted to the system as a set and the database system is then tasked with generating shared query plans that, when executed, provide the results for the entire set of queries.

Another common approach is performing MQO in realtime contexts. Since all the information about queries is not available at the outset, realtime MQO techniques usually involve generating and caching materialized views during query execution and using these views when executing subsequent queries.

### Limitations of MQO

Both batched and realtime methods of MQO suffer from some problems. The performance of batched MQO techniques, for example, depends on the time taken to batch queries [Sel88]. Longer batching times result in delayed responses; on the one hand, to ensure good efficiency, the batch needs to be as large as possible so that the chances for generating a shared query plan increase. On the other hand, for getting quick responses, the batch needs to be as small as possible so that it is executed quickly. Finding a balance between these two opposing requirements is dependent on the properties of queries, which may not be known beforehand, possibly resulting in sub-optimal performance when using batching.

Additionally, in batched MQO techniques information is generally not persisted. Shared query plans, after being generated and executed for a set of queries, are discarded. This is similar to the problem of isolation that we observe in traditional database systems, with the minor difference being that resources are isolated with respect to batches rather than individual queries.

The problem with realtime MQO is that deciding what resources to cache and what to disregard requires *a priori* information about queries, which is not available when they occur randomly, as they do in realtime systems. Deciding to store ineffective information can lead to a performance that is worse even than sequential execution.

### Hybrid MQO Technique

To overcome these limitations, we propose a hybrid MQO approach that combines the characteristics of both batched and realtime processing based techniques, and as a result can process queries of both natures.

Initially, queries are batched, wherein a composite query is created from a set of queries. This composite query contains the results of all its constituents. Then, these composite queries are evaluated against a cache of materialized views to find suitable views for substitution. Materialized view substitution refers to the process of altering a query to retrieve results from the materialized view instead of the database. Our hybrid MQO technique is discussed, in more detail, in Chapter 3.

### Contribution of This Thesis

In this thesis, we design and implement a hybrid MQO approach that can work in both batched and realtime contexts. We also examine the performance of our hybrid MQO approach with respect to sequential, batched, and realtime execution of the queries by evaluating it under many different situations resembling different practical settings. Specifically, we consider the following aspects when designing and evaluating our hybrid method:

- The viability of using our hybrid method for analytical workloads.
- Performance as compared to Shared Sub-Expressions (SSE), Materialized View Reuse (MVR), and sequential execution.
- The effect of different database operators on the performance of our hybrid method.
- The impact of cache size on the performance of our hybrid method.
- Performance evaluation of our hybrid method for different levels of query similarities, showing, in the best-case, a performance benefit of 2x over sequential execution.

### Structure

This thesis is structured as follows. In Chapter 2, we begin our discussion with a brief overview of the query execution process. We will also consider, in detail, how common MQO techniques function. A comprehensive description of our hybrid MQO technique is then presented in Chapter 3. Comprehensive evaluation of our hybrid method is carried out in Chapter 4, with comparisons describing the relative performances of our hybrid method with sequential, batched, and realtime execution. We then take a look at the relevant literature in Chapter 5. We provide our conclusion in Chapter 6 and finally, Chapter 7 presents avenues for further improvements and analyses.

# 2 Background

Database systems handle query processing in stages. Even though the exact nomenclature of these stages varies between different database implementations [EN16, GMUW09], the functions are the same. Generally, there are three stages a query goes through before the result is given back to the user: Parsing, Optimization, and Execution.

In this chapter, we will discuss briefly the significant stages of query processing, both in the context of traditional as well as multiple query processing. Additionally, we will examine how materialized views can be used to *derive* query results in Section 2.2. Finally, in Section 2.3 we will take a deeper look into Multi-Query Optimization (MQO). Specifically, we will study the common techniques used for optimizing multiple queries in contexts where shared processing has certain advantages.

## 2.1 Query Processing

Query processing refers to the method of extracting relevant results from the database. Figure 2.1 depicts the stages that queries pass through in a database system. As we have discussed earlier, there are three distinct stages that process queries before the result is obtained. In this section we will consider the role of each stage, starting with Parsing.

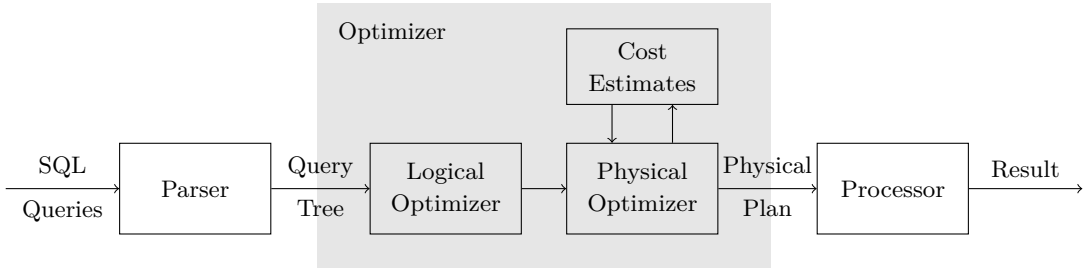



Figure 2.1: Block diagram of a database system

### 2.1.1 Parse

The first step in a database's query processing pipeline is parsing a query. Parsing can further be divided into two sub-steps. First, the database checks the syntactical accuracy of the query. This includes checking whether appropriate data types and tokens are used, proper tables are joined with suitable join predicates, proper columns are aggregated upon, among other checks. Then, a data structure, usually a tree or a graph, is constructed from the query string so that further processing and manipulation is easier. An illustration of a query tree for the below given query $Q$ is shown in Figure 2.2.

```
Q: SELECT o_totalprice, c_name
   FROM orders JOIN customer ON o_custkey = c_custkey
   WHERE o_totalprice < 10000 AND c_acctbal < 50000
```

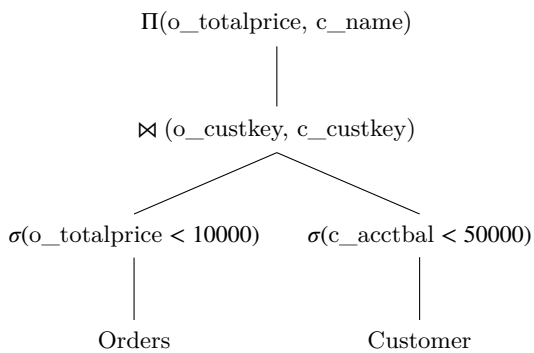



Figure 2.2: Query tree

### 2.1.2 Optimization

After a query tree is generated, a logical plan is created, which is a high-level description of what the query must undertake. For instance, if the SQL query is `SELECT * FROM orders`, then the logical plan asserts that the table *orders* must be scanned. However, it does not define which implementation of the scan algorithm should be used.

The goal of the optimizer is to convert a logical plan into a physical plan — one that assigns concrete implementations to the operators. This is achieved in two ways:

1. Heuristic optimization
2. Cost-based optimization

#### Heuristic optimization

In this step, rule-based transformations of a logical plan are performed. The rules are specified ahead of time and ensure relational equivalency. The aim of applying

transformation rules is to reduce the number of intermediate tuples that are generated by reordering the operations in the query tree[EN16].

A simple yet important heuristic rule is the application of `SELECT` and `PROJECT` operations before `JOIN`, as joins usually result in multiplicative outputs of the given relations. Because of this reason, the optimizer pushes selections and projections to as close as possible to leaf nodes. Some other common rules include predicate push-downs, selection-projection commutativity, cascading selections, commuting set operations, and so on. [EN16].

These rules are mathematical in nature and as such are independent of the specific characteristics that the stored data may exhibit. A more narrower scope of optimization is utilized in the second step of query optimization: cost-based optimization, where statistical information extracted from the data instructs the optimization process.

### Cost-based optimization

A query optimizer cannot merely rely on a heuristic planner for generating efficient query plans, except perhaps for the simplest of queries. In addition to rule-based transformations of the query, a cost-based approach is also commonly utilized. In such a method, equivalent query plans are generated by traversing the available breadth of the solution space to find a plan with an optimal cost [EN16]. The search space is described by factors such as the operators present in the logical plan, their availability and applicability, and statistical information about the underlying data.

Consider the possibilities for arriving at the result of a natural join of three relations, say $\boldsymbol{A} \bowtie (\boldsymbol{B} \bowtie \boldsymbol{C})$. Because the natural join operator is associative, the relations $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ can be shifted without any difference in the output. This is illustrated in Figure 2.3. For the given three relations, there are 12 possible join orderings that produce the same results. Based on the statistical properties of the relations, some join orderings will be more efficient than others. For instance, if the selectivity of $\boldsymbol{A} \bowtie \boldsymbol{B}$ is very low, then a large number of tuples will be present in the intermediate relation if this join is processed first, which puts the plan at a disadvantage.

Determining efficient query plans is achieved with the help of a cost estimator (cost model). The cost estimator determines the cost of the particular plan, i.e. a quantitative measure that describes what executing that particular query demands. The cost estimator is one of the most important parts of the optimizer; if it is incorrectly configured, then the database arrives at non-optimal plans, which adversely affect its performance. Despite this requirement, the cost model cannot be completely and invariably accurate. Statistical information about data is not a substitute for the data itself.

Additionally, the entire solution space cannot be traversed by the optimizer because the number of possible permutations of the operators is exponential. Due to this

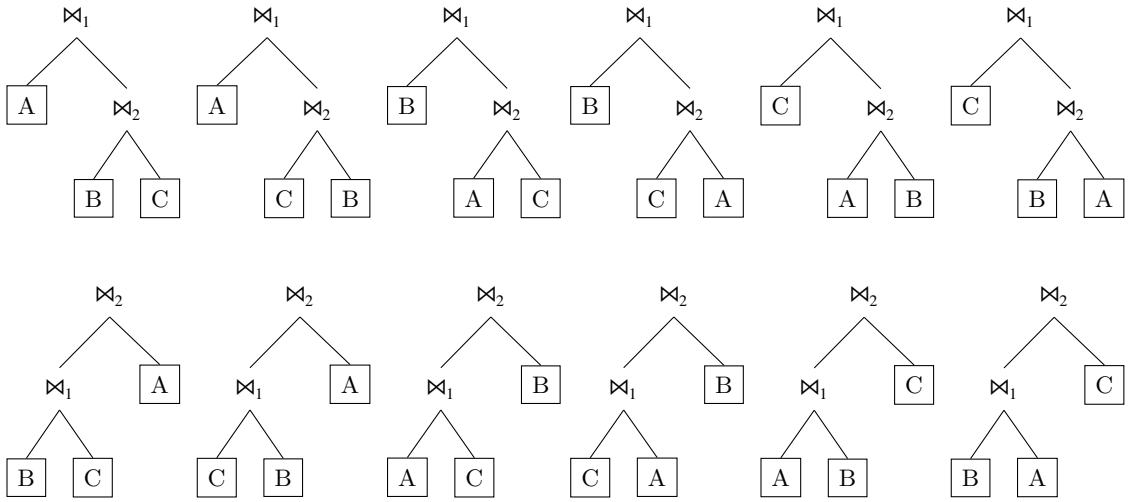



Figure 2.3: Possible join orderings for three relations

reason, the optimizer is tasked with finding plans that are optimal to a certain extent. Nonetheless, modern database optimizers are very efficient at finding query plans.

The cost model takes into account both the internal aspects of the database system — such as the data distribution – and the external environmental properties — such as the estimated memory usage — to arrive at the cost for a query plan. While every database system differs in the exact implementation of the cost model, commonly considered factors are

- Choice of operator algorithms

  Selection of which physical implementation of an operator to use plays a crucial role in cost estimations. Consider again, for instance, the case of natural joins. Joins can be implemented using algorithms for nested-loop join, or sort-merge join, or hash join [ME92]. If we have prior knowledge that a certain relation is already sorted, then sort-merge join would provide the best performance by far. And if, perhaps, we know that the relations are very small then a nested-loop join would probably be faster than a hash join because the latter requires special data structures to be created and populated which might not be worthwhile for a relation with few rows. Therefore, the planner needs to be aware of the various available implementations for operators and should be able to judiciously use the appropriate ones.

- Resource utilization

  How heavy a query is (in terms of its resource footprint) also affects the cost of a plan. This includes the estimated CPU load that the system will be put

under, and the amount of memory that will be used [Cha98]. If the database system is disk-based, then the I/O cost is also included. Additionally, for distributed systems spread out over a network, a further addition of the network communication costs also has to be done.

- Inherent properties of the data

  Database systems maintain statistical summaries of the stored data. These include basic information about tables such as its cardinality, the number of unique values in a column, formulas to calculate selectivities for various columns, and so on.

  For larger systems, databases also include histograms for columns, which provide a distinct number of bins that the column is segregated into, making the task of calculating statistical properties for that column simpler. Unfortunately, while histograms provide good information about individual columns, for correlated columns they are not useful. In that case, basic statistical information along with histograms provide a good approximation for generating relatively accurate plan costs [Cha98].

The conversion of a logical plan into a physical plan is not one-to-one; a single logical operator does not necessarily map to a single physical operator. Logical operators can split to form multiple physical operators, and multiple logical operators can coalesce into a single physical operator.

After a sufficiently optimal physical plan is selected, the optimizer passes on the plan to the query executor.

### 2.1.3 Execution

In this stage, the selected query plan is initially converted into code that can be executed by the query processor. The actual procedure for query execution, defined by the query execution model, can be designed in various ways. The three most prominent query execution models are

- Tuple at a time / Iterator model

  In this execution mode, each operator implements a `next()` function that processes and returns a tuple — or nothing if no output exists. The advantage of this model is that since each tuple is processed exclusively, the operations can be pipelined to achieve a performance gain. A visual representation of the Iterator model is shown in Figure 2.4a.

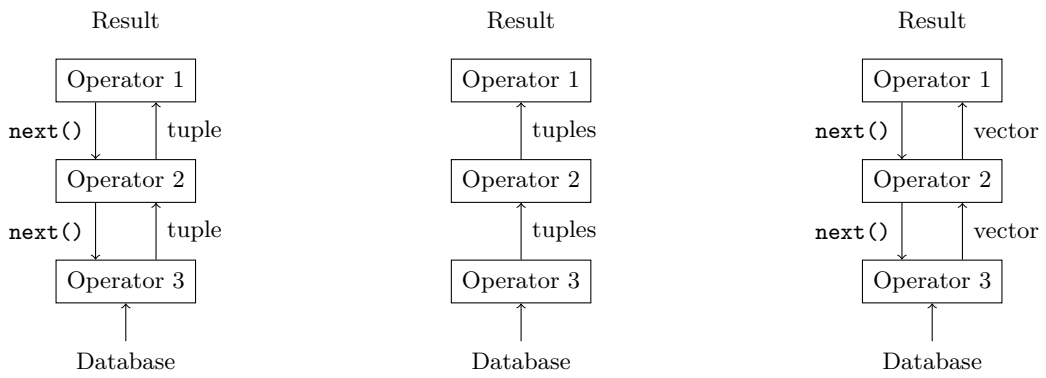
- Operator at a time / Materialization model

  As the name suggests, each operator processes the entirety of its input before the execution moves to the next operator, as depicted in Figure 2.4b. Because every

operator processes its complete input, we have access to the intermediate results that are generated. Using these intermediate results forms the foundation for some of the MQO techniques, as will be discussed in further sections.

- Vector at a time model

  As in the Iterator model, each operator in a Vector at a time model also implements a `next()` function. But, as shown in Figure 2.4c, instead of returning just a single tuple, operators emit a vector (batch) of tuples. The size of this batch varies based on query or hardware properties. Like the Iterator model, intermediate results are not generated in this model too because of the distributed nature of tuple processing.




(a) Tuple at a time (b) Operator at a time (c) Vector at a time

Figure 2.4: Query execution models

We have thus far been concerned with how a single query is executed in a conventional database system. In summary, the query is parsed to detect any syntactical errors. Then an alternative representation of the query is generated, which is usually a tree or a graph. A logical plan that outlines the operations to be performed for obtaining the result is then generated from the tree. The final physical plan is arrived at through rule-based and cost-based optimizations of the logical plan. This physical plan is then converted into code that is executed by the database processor. Finally, the results are provided to the user.

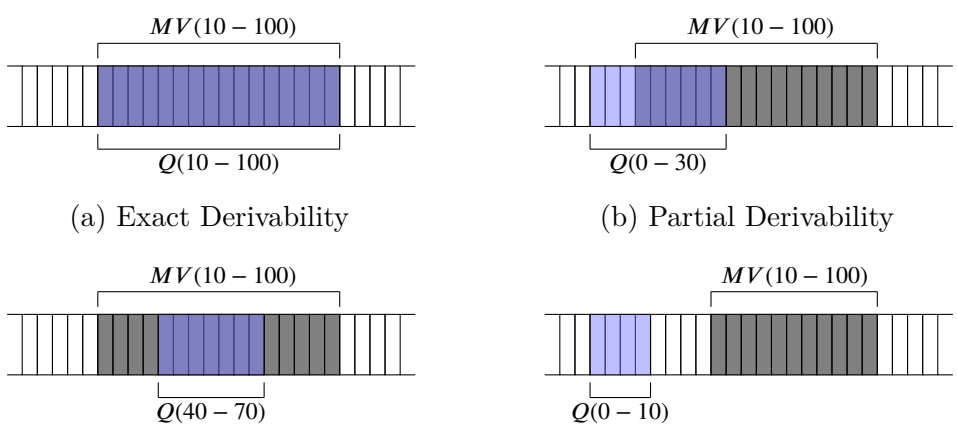
The information gathered when processing a query — such as the statistics, and the table and column information — is usually cached temporarily in order to easily and quickly execute any subsequent queries of similar nature. Unfortunately, these measures are not directed towards inter-query processing but instead towards faster plan generation for future queries; the difference being that queries are still executed singly. A relatively simple approach to introducing an element of MQO to conventional query processing without much alteration of the query planning part can be achieved by using materialized views.

## 2.2 Materialized Views

Although most database systems cache some intermediate information that is generated when processing queries, a query processor encountering a new query will still retrieve the results from the database from scratch. This misuse of system resources can be overcome using *materialized views*.

A materialized view is a table representing the query and its results. In other words, it is a copy of the query results residing in memory. Materialized views can be stored temporarily, for instance in cases where a set of similar queries need to be executed, after which the materialized view can be removed. Or materialized views can be stored permanently. Since materialized views are stored in the main memory, it is much cheaper for the optimizer to read suitable data from the materialized view (provided that it is possible to do so) than to retrieve the data from the disk. This is especially true for heavier operators such as joins, aggregates, and sorts [EN16].

The property of deriving query results from a materialized view is called Derivability [DBCK17, RSSB00]. There are four general types of derivabilities: *None*, *Exact*, *Partial*, and *Subsuming*. If we consider a query $\boldsymbol{Q}$ and a pre-existing materialized view $\boldsymbol{MV}$ containing rows 10-100, then we can say that the type of derivability is




(a) Exact Derivability (b) Partial Derivability

(c) Subsuming Derivability (d) Non-derivable Query

Figure 2.5: Types of derivabilities

- **Exact**, if $\boldsymbol{Q}$ requires rows $(10-100)$ since these are precisely the ones present in the view $\boldsymbol{MV}$, as shown in Figure 2.5a.
- **Partial**, if only a subset of results that $\boldsymbol{Q}$ requires are present in the view $\boldsymbol{MV}$. As depicted in Figure 2.5b $\boldsymbol{Q}$ requiring rows $(0-30)$ is an example of this case.
- **Subsuming**, if $\boldsymbol{Q}$ requires rows $(40-70)$ as shown in Figure 2.5c, since $\boldsymbol{MV}$ contains a superset of required results.

- **None**, if no results for the query $\boldsymbol{Q}$ are present in the view $\boldsymbol{MV}$, as shown in Figure 2.5d.

In this section we have examined how materialized views can be used for extracting the results of queries and how, using the concept of derivability, we can quantify the degree to which a materialized view is capable of providing results to queries. In the next section, we will take a more broader look at MQO, with deeper examinations of two important types of techniques that are used in executing queries mutually.

## 2.3 Multi-Query Optimization (MQO)

There are many applications where a database system is presented with multiple queries to be processed concurrently. For example, in multi-user applications, many users can submit queries to the database simultaneously. Traditional database execution, as we have seen so far, executes these queries in isolation. This approach does not pose a problem when the queries are dissimilar to each other, as sharing execution between such queries would make the performance indifferent at best and degraded at worst.

But if the queries share a non-negligible amount of similarities, perhaps in accessing the same tables, joining over the same columns, or filtering over similar predicates, then the traditional approach tends to redo former optimizations all over again, even though there is a possibility for resources to be reused. Furthermore, this problem is compounded when the queries produce very large results, as is the case for *selection-only* Online Analytical Processing (OLAP) systems [GAK12].

Multi-query Optimization solves this problem by processing similar queries that occur in a certain timeframe efficiently. Here, the goal is to produce an execution strategy that is efficient in processing multiple queries irrespective of its performance for individual queries [Sel88]. In MQO, queries can be processed in two ways: as a batch or in realtime.

### 2.3.1 Batched MQO

In batched MQO [GMAK14, MCM19, SLZ12], multiple queries are bundled, which the database system then executes by forming a single query execution plan. In particular, we create a batch from all the queries that occur in a certain timeframe $\tau$, as shown in Figure 2.6.

An advantage of batched processing of queries is that the information required by the optimizer — the types of queries and their operators and predicates — to make informed decisions is available upfront, making the process of optimization more sound.

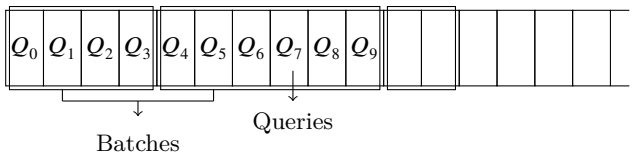



Figure 2.6: Batched MQO

An obvious problem with this approach is that the time taken to create batches, $\tau$, is inversely related to the optimization performance. Consider the case where batches are created within a very short timeframe. These batches would yield a very short response time — because there are fewer queries to process. But the extent of optimization would degrade because fewer queries indicate fewer similarities across queries, and by extension fewer chances for optimization. This is a trade-off problem: batching queries over a larger timeframe ensures good optimization at the expense of response time. And batching queries quickly ensures a good response time but poor optimization. Unfortunately, there is no ubiquitous solution to what the batching time should be, primarily because this is context dependant. The frequency and the types of queries that occur, and the types of operators that the queries contain play a role in determining the ideal batching time.

Instead of combining several queries to form a batch, an alternative approach is to split a single query into many sub-queries, each of which can be considered as a part of a batch. This is advantageous in situations where there is a lot of inherent complexity in every query or if the queries contain several similar sub-expressions.

The technique primarily used in batched MQO is Shared Sub-Expressions (SSE). As the name implies, in this technique common sub-expressions are identified among queries of the set that are then executed only once. The result of the expressions are propagated to all the required queries. In the literature, there are other methods that achieve batched MQO, such as sharing the execution of operators as employed by SharedDB [GAK12], or pipelining the required intermediate results to queries without any processing on the systems' part as suggested by Dalvi et al. [DSRS03], but by far the most widely used technique is SSE [MCM19, SLZ12, RSSB00]. Because of this reason, and the fact that our hybrid approach is closer to SSE than to the other techniques, we limit our discussion of batched MQO to SSE.

### Shared Sub-expressions (SSE)

In Shared Sub-Expressions (SSE), possible common expressions are initially identified among a set of queries. The identified sub-expressions are executed in parallel, with their results being provided to the queries that require them. Specifically, a shared execution plan is created that includes covering expressions derived from combining the sub-expressions.

A simple example of SSE is shown below. Consider queries $Q_1$ and $Q_2$ as part of the same set of queries. We can see from the queries that the common sub-expressions would be the ones concerning `shipdate` and `discount`.

$Q_1$:
```
SELECT discount, quantity
  FROM lineitem
  WHERE shipdate > 1994-01-01
    AND shipdate < 1994-06-02
    AND discount > 0.02
    AND quantity > 32
```

$Q_2$:
```
SELECT tax, shipdate
  FROM lineitem
  WHERE shipdate > 1994-04-01
    AND tax < 0.02
    AND discount > 0.04
```

If we were to batch these queries together, the resulting query would look like the following. We can see that the filtering on dates is reduced to accommodate the results for both the queries, whereas the filtering on quantity is removed altogether for the same reason. Essentially, we have generated a query that *covers* the result of both $Q_1$ and $Q_2$.

$Q_3$:
```
SELECT discount, quantity, tax, shipdate
  FROM lineitem
  WHERE shipdate > 1994-01-01
    AND discount > 0.02
```

Naturally, as the number of queries in a set increases, the possibility of finding suitable common sub-expressions also increases. This ties in with the problem of delayed response, as we have discussed in the preceding part: we need to balance the response time with the batching time. Asymmetry in these variables can produce non-optimal performance.

A challenging aspect to consider when implementing SSE is finding, from a set of queries, the sub-expressions that are common. Michiardi et al. [MCM19] and Silva et al. [SLZ12] use fingerprinting to identify common sub-expressions, which involves comparing the hashes of expressions. The sub-expressions are considered similar if the hashes are same.

This presents a unique difficulty: using hash functions to determine expressions that are not just exactly but also relatively similar. The definition of relativity may depend on the implemented approach — some approaches consider expressions with exactly same predicates as common whereas other approaches consider common the expressions that use similar columns. In addition to helping determine common sub-expressions, the latter method of designing the hash function to be invariant of operands allows for the possibility of derivability.

Instead of using fingerprinting to identify common sub-expressions, Kathuria and Sudarshan [KS17] use a heuristic approach, where simple and intuitive pre-defined rules are utilized.

In conclusion, batched MQO, and by extension SSE, has certain advantages and limitations. These advantages and limitations are in contrast with another type of MQO: relatime processing.

### 2.3.2 Realtime MQO

In realtime processing, the queries are submitted to the database system as they occur [PJ14]. The information stored by the system for optimizing queries can be viewed as a growing window, as shown in Figure 2.7. This type of processing is more in line with how real-world systems receive queries.

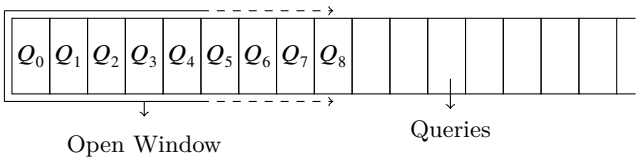



Figure 2.7: Realtime MQO

The biggest advantage that realtime processing offers over batched processing is its promptness in delivering results because each query is analysed for its optimization potential separately instead of in a batch.

Even though realtime processing solves the problem of delayed responses posed by batched processing, it has few problems of its own. Firstly, when compared with batched MQO, processing queries in realtime and separately indicates that there are fewer avenues for optimization — as complete information about queries is not available at the outset. Secondly, the determination of whether and to what extent a query is optimizable, and the process of optimization should be quick so as to offset the gains against sequential execution.

Commonly, there are two main techniques for realtime MQO: Intermediate Results Reuse (IRR) and Materialized View Reuse (MVR). In the former, the intermediate results that are generated during query execution are stored and reused when processing operators of subsequent queries. This method has limited applicability because not every execution model generates Intermediate Results (IR). As we have seen in Section 2.1.3, different databases use different query execution models. Of the three major execution models, only Operator-at-a-time model generates IR. Even though not all databases use a suitable model for the application of IRR, the literature on this type of MQO is extensive.

Another method for optimizing multiple queries in realtime contexts is MVR. As the name implies, materialized views based on the current queryload are cached, which are then used in aiding execution of suitable subsequent queries. Specifically, instead of retrieving data from the database, the materialized view is used. An advantage of MVR over IRR is its universality. Since generating materialized views does not require any special constraints, this method is applicable on a wider range of database systems. As a consequence of this, and its importance in realizing the hybrid method of MQO, we will discuss MVR in more detail in the following section.

### Materialized View Reuse (MVR)

As we have seen in the previous section, in Materialized View Reuse (MVR) we need to generate and cache suitable materialized views, which will then be matched against successive queries for their reusability. Consider the example shown below. $\boldsymbol{MV}$ represents the cached materialized view and $\boldsymbol{Q}$ the query that the database system has to execute.

$\boldsymbol{MV}$:
```
SELECT discount, quantity
FROM lineitem
WHERE discount > 0.02
   AND quantity > 32
```

$\boldsymbol{Q}$:
```
SELECT quantity
FROM lineitem
WHERE discount > 0.04
    AND quantity > 35
```

We can see that the results of $\boldsymbol{Q}$ can be derived from the materialized view since the constraints that define $\boldsymbol{Q}$ are a subset of the constraints that define $\boldsymbol{MV}$. Thus, the query $\boldsymbol{Q}$ can be rewritten to use $\boldsymbol{MV}$ as

$\boldsymbol{Q}_{\boldsymbol{MV}}$:
```
SELECT quantity
FROM MV
WHERE discount > 0.04
   AND quantity > 35
```

While this is an elementary example demonstrating the basic principle behind MVR, it can nonetheless be used for queries with a higher degree of complexity, given that the data present in the materialized view is relevant. The concept of derivability, as discussed in Section 2.2, is critical in determining if materialized view reuse is possible.

There are some inherent challenges in such a system, such as the selection of materialized views for caching and substitution, and their storage. Gosain and Sachdeva [GS17] have performed a comprehensive survey on the best approaches for selection of materialized views in a high-volume query processing system. In some applications, perhaps some meta-information about queries — like their frequency and types of operators — is known. Although generally, the nature of realtime processing is such that *a priori* information about queries is not available. In this case, there are few

choices for selecting which materialized view to cache. Some approaches, like Perez and Jermaine [PJ14], prefer extracting information from the historical queryload to determine the viability of a view being utilized in the future.

Furthermore, a second issue is the physical and logical storage of the materialized views. The time taken to access the stored materialized views must be low, and so must the selection algorithm for determining whether a materialized view is suitable for reuse. Inefficient implementation of these issues can be harmful to the performance of the system.

In regards to the former matter of efficient physical storage of the materialized views, there are essentially three options available: the CPU cache, the RAM, or the disk. Although the CPU cache offers near-instant access times, it is extremely limited in size. On the other end of the spectrum, the disk offers a large storage capacity albeit at the cost of slower I/O. Many approaches favor the RAM, which offers a reasonable compromise between the two extremities.

For the latter issue concerning the efficiency of the selection algorithm, much of the literature favors using a hash table. In the often occurring case that many materialized views need to be cached, an appropriate hash function can be chosen that provides good efficiency. Similar to what we have seen in Section 2.3.1, the chosen hash function must be such that it allows for derivability.

Sometimes it may also be the case that multiple materialized views are selected as potential candidates for reuse, in which case a single view needs to be picked based on some criteria. Some approaches use a cost-based utilitarian method where every materialized view is assigned a *value*, perhaps based on how frequently it was used in the past, and the view with the highest *value* is picked. Yet other approaches use a greedy measure where the best possible view for the current scenario is selected regardless of its impact on other factors.

We have so far studied the main techniques that encompass MQO. To summarize, multiple queries can be processed by any system in one of two ways: as a batch or in realtime. Batched processing provides some benefits such as providing complete information about queries, but at the cost of slower response times. Shared Sub-Expressions (SSE), a batched MQO technique identifies common sub-expressions from queries and then evaluates these sub-expressions in a common context.

On the contrary, realtime processing provides swift responses but suffers from a shortage of information. Materialized View Reuse (MVR), a realtime MQO technique caches and reuses materialized views to facilitate optimization of many queries.

Techniques belonging to both approaches have their benefits and drawbacks, which are in opposition. On the one hand we have batched processing (SSE) where global shared query plans are generated and executed but no information is persisted. And

on the other hand we have realtime processing (MVR), storing materialized views but not creating shared execution plans.

We propose, in this thesis, a hybrid MQO technique that places itself in the midst of these two approaches. In this way, we can leverage the benefits of both the techniques while minimizing the limitations. In the next chapter, we will discuss the core concepts that form the basis for the hybrid MQO technique.

# 3 Hybrid Multi-Query Optimization

In the previous chapter, we have looked at how queries are generally executed and what Multi-Query Optimization (MQO) involves. We have studied the constituent parts of a query processing system, namely the parser, the optimizer, and the executor. And we have discussed the idea behind MQO, with a focus on the two main ways of processing multiple queries.

With that understanding, we conceive the working of a hybrid MQO system. We will begin with an overview of the system and then move on to the distinct components that such a system consists of.

## 3.1 Overview

In addition to the components of a traditional database system, one that processes multiple queries needs to have, depending on the type of MQO, certain additions such as a query batcher, a substituter, and a cache.

In the case of Shared Sub-Expressions (SSE), a mechanism for batching queries and generating shared plans is needed. Similarly, for Materialized View Reuse (MVR), we need to store materialized views and an efficient way of using them to facilitate optimization.

In the hybrid MQO method we propose, as we are combining both batch and realtime approaches, we need a *Query Batcher* and a *Substituter*. The batcher receives a set of queries which it then condenses into a single query plan. We detail the batching mechanism in Section 3.2.

The second main component is the substituter. This part is responsible for storing and matching materialized views with queries that are to be executed. To this end, we also need to store a cache of historical materialized views. The substituter picks suitable materialized views from the cache for reuse and alters the existing query plan such that the materialized view is used instead of reading from the database.

The structure of the hybrid system is shown in Figure 3.1. The blocks colored in gray belong to a conventional database system and those in white are exclusive to the MQO system.

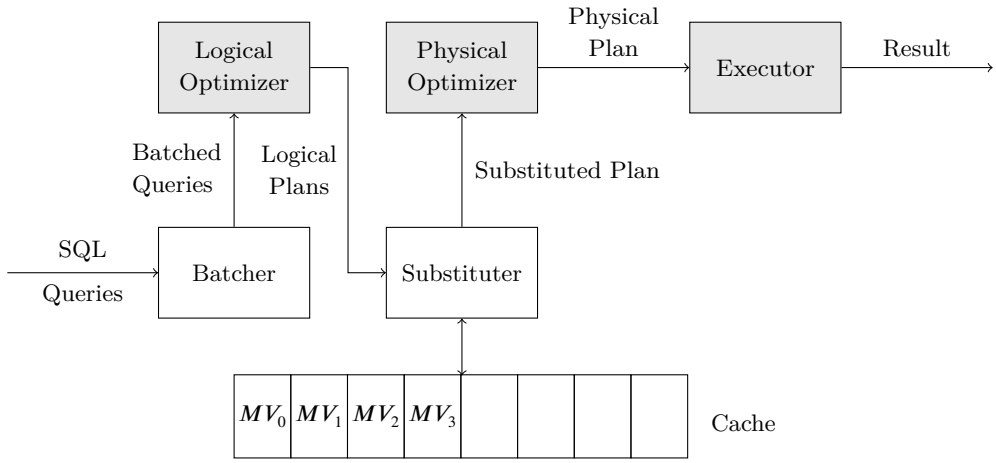



Figure 3.1: Block diagram of our hybrid MQO system

Initially, a set of queries are parsed and then submitted to the batcher. Generation of queries is not considered as central part of our hybrid MQO system as it does not contribute towards the performance of the system. When the generated queries are parsed, as discussed in Section 2.1.1, we ensure that the queries are syntactically correct and then create a logical representation for every query string; a tree, for example.

The parsed query set is then submitted to the batcher, where eligible queries are batched to form another set. The list of batched queries goes through the process of logical optimization, similar to one in a conventional query execution process. In this phase, the queries are converted from query trees to a implementation agnostic query plan.

After logical optimization, the queries are optimized in the manner of techniques presented in Section 2.3.2. This involves finding and picking suitable materialized views from cache that can aid in the query's execution. This is achieved through derivability, as we have seen in Section 2.2. Derivability can be defined as being one of four types: none, exact, subsuming, or partial derivability. If no suitable view that can provide its results to the query is found, then we create a materialized view from the query and cache it for future use. Instead, if a materialized view is found that we can use to derive the results of the query from, we alter the query plan to use the materialized view for the relevant parts of the query's execution.

For materialized view substitution to function efficiently, we need the cache to be appropriately sized. Since the size of the cache dictates how long as well as how many

materialized views can be stored, it plays a crucial role in the performance of the system. If the cache is too small, then the number and the lifetime of the views would be inadequate, affecting negatively the performance of the system. On the other hand, if the cache is sized too large, then there is a possibility for resources to remain unused. Furthermore, as a consequence of storing a large number of materialized views, we must also, regardless of their relevancy, check every view for its substitution capability, which can cause delays in the response time. In our case, we have used a map to store our materialized views

The next step, namely physical optimization, is generally the same here as in a conventional database system. As discussed in Sections 2.1.2 and 2.1.3, we attach concrete implementations of the various operators present in the logical plan to generate a physical plan that is then executed by the query processor.

In the following pages of this chapter, we will look at the three main inter-related activities of our hybrid MQO method that enable it to work efficiently: how queries are batched, how materialized views are substituted, and how the cache functions. Finally, we will look at how these three parts are combined to form the complete hybrid MQO system.

## 3.2 Query Batching

The first step that takes place in the hybrid MQO system when a set of queries is submitted is that they are batched. In our implementation, we batch queries based on the similarities of queries' `WHERE` clauses. Currently, only the most commonly used operators such as selections, projections, natural joins, and aggregations are taken into consideration when batching. Nevertheless, other SQL operators can also be incorporated into the batching algorithm.

Additionally, there is also the consideration of datatypes. Different datatypes require different handling mechanisms during batching and as such we have taken the same approach for datatypes as we have for operators. That is, we have implemented batching for the most commonly used datatypes: integers, floating point values, strings, and dates. Nonetheless, although we have gone with this approach there is no reason why this cannot be extended to any arbitrary datatype.

We focus primarily on the generation of a composite query plan from a given set of queries. Generally, this is achieved by accumulating queries that occur in a certain timeframe ($\tau$). As this value is dependent on the nature of queries, we assume that sets of queries are readily available and place a greater emphasis on how a composite query is created.

The general procedure for batching a set of queries is as follows:

1. Group queries based on their eligibility for being batched together.
2. Exclude any group that has fewer than 2 queries, as a single query cannot be batched.
3. For each remaining group, perform the following operations
   3.1. Convert the `WHERE` clauses of the queries to their respective Conjunctive Normal Forms (CNFs).
   3.2. Combine the `WHERE` clauses using a logical `OR` to generate a composite `WHERE` clause.
   3.3. Expand the composite `WHERE` clause using algebraic rules.
   3.4. Simplify the composite `WHERE` clause by removing redundant and irrelevant predicates.
   3.5. Reconstruct selections for the composite query from all its constituent parts.
   3.6. Combine the composite selections with the composite `WHERE` clause to obtain the composite query.

In the following parts, we will study each step in detail, with suitable examples, and then consider how, by the end, we can efficiently generate composite queries. When a set of queries is to be batched, we must determine which queries from the set are appropriate to be batched together, and which are not. This is conducted by defining a criteria for batching.

### 3.2.1 Batch Criteria

In our hybrid MQO method, we have defined queries that access the same tables as being eligible for batching. This is illustrated in the list of queries shown below. We can see that $Q_1$, $Q_2$, and $Q_5$ access the same table: *lineitem*. Similarly, $Q_4$ and $Q_6$ scan the *supplier* table. $Q_3$ on the other hand, is a query which accesses the *partsupp* table, which no other query in our list does. So we can infer from our criteria that queries $Q_1$, $Q_2$, and $Q_5$ can be batched and thus can be considered as a group. There is a similar case to be made for queries $Q_4$ and $Q_6$. Finally, $Q_3$ is not batched because it is singular in terms of what tables it accesses.

$Q_1$:
```
SELECT l_quantity, l_extendedprice, l_shipdate, l_tax, l_discount
    FROM lineitem
    WHERE l_discount > 0.06 AND l_quantity > 15
```

$Q_2$:
```
SELECT count(*), l_quantity, l_discount
    FROM lineitem
    WHERE l_discount > 0.08 OR (l_quantity > 23 AND l_tax < 0.02)
    GROUP BY l_quantity, l_discount
```

$Q_3$:
```
SELECT ps_suppkey, AVG(ps_availqty)
    FROM partsupp
    WHERE ps_availqty < 68 AND ps_partkey < 6421
    GROUP BY ps_suppkey
```

$Q_4$:
```
SELECT s_name, s_suppkey, s_acctbal, s_nationkey, n_nationkey
    FROM supplier JOIN nation on s_nationkey = n_nationkey
    WHERE s_suppkey < 5000 AND s_acctbal < 9987
```

$Q_5$:
```
SELECT l_quantity, avg(l_discount)
    FROM lineitem
    WHERE l_tax < 0.07 AND l_discount > 0.02
    GROUP BY l_quantity
```

$Q_6$:
```
SELECT s_acctbal, n_name
    FROM supplier JOIN nation on s_nationkey = n_nationkey
    WHERE s_suppkey < 5000 AND s_acctbal < 9987
```

In this way, we pick what queries can be considered as candidates for batching and group them together. Next, for each group of queries we have to generate a single query plan such that its results contain the results of all its constituents.

### 3.2.2 Generating a Composite Clause

Initially, the `WHERE` clauses from all the queries are isolated. Before we can combine the clauses from all the queries, we must ensure that the clauses are in the canonical form, ignoring which could lead us to incorrect predicates when combining them. This is why we convert every clause to its equivalent Conjunctive Normal Form (CNF) before proceeding any further.

#### Converting to CNF

Since the `WHERE` clause for each query can contain a combination of conjunctive (using `AND`) or disjunctive (using `OR`) predicates, we convert the clauses into its equivalent CNF. An example of a `WHERE` clause in its original and the normal form for the query $Q_2$ from our previous example is shown below.

```
// Original form
WHERE l_discount > 0.08 OR (l_quantity > 23 AND l_tax < 0.02)
```

```
// Conjunctive normal form
WHERE (l_discount > 0.08 OR l_quantity > 23)
      AND
      (l_discount > 0.08 OR l_tax < 0.02)
```

After CNFs for all the `WHERE` clauses in a group are generated, the next step involves creating a composite `WHERE` clause by combining all the individual CNF clauses with a logical `OR`.

We use the `OR` because we require the results of both the queries. If instead we use `AND`, we would only produce the intersectional results of both queries.

### Combining Clauses

A simple yet lengthy composite `WHERE` clause can be generated by simply combining all the constituent clauses with `OR`. The composite clause may be lengthy because it may contain redundant information. Since the constituent queries can be rather long, a simple `OR`ing of the clauses can often end up with very long and confusing composite `WHERE` clause. An example of such a situation is shown below. Consider once again the example queries established above. The conjunctive normal forms of the where clauses for queries $Q_1$ and $Q_2$ are given below.

```
Q1: l_discount > 0.06 AND l_quantity > 15

Q2: (l_discount > 0.08 OR l_quantity > 23)
      AND
     (l_discount > 0.08 OR l_tax < 0.02)
```

The combined query, resulting from `OR`ing these two expressions, would then be

```
Q: (l_discount > 0.06 AND l_quantity > 15)
        OR
    ((l_discount > 0.08 OR l_quantity > 23)
            AND
    (l_discount > 0.08 OR l_tax < 0.02))
```

This is a boolean expression of the form $A \vee (B \wedge C)$ which can be expanded, using the rules of boolean algebra, to $(A \vee B) \wedge (A \vee C)$. In the case of our where clause, it would give us

```
Q: (l_discount > 0.06 AND l_quantity > 15)
        OR
     (l_discount > 0.08 OR l_quantity > 23)

    AND

    (l_discount > 0.06 AND l_quantity > 15)
       OR
    (l_discount > 0.08 OR l_tax < 0.02)
```

Going one step further and again expanding the above expression to only be one level deep, we get

```
Q: (l_discount > 0.06  OR l_discount > 0.08 OR l_quantity > 23)
        AND
     (l_quantity > 15 OR l_discount > 0.08 OR l_quantity > 23)
        AND
    (l_discount > 0.06 l_discount > 0.08 OR l_tax < 0.02)
        AND
    (l_quantity > 15 OR l_discount > 0.08 OR l_tax < 0.02)
```

Although this clause would provide correct results, it is not ideal to be processed further due to its length, which increases exponentially with the lengths of its constituents. This problem can be overcome by simplifying the composite clause.

### 3.2.3 Simplifying the Composite Clause

Simplification of the generated composite `WHERE` clause is done, in our hybrid MQO system, in two ways. Firstly, predicate groups are simplified using pre-defined rules. This simplification ensures that irrelevant or incompatible predicate groups are removed so that such groups are not processed in the next step, which creates covering predicates.

#### Predicate Group Simplification

In the previous step, we have reached the CNF form for our composite `WHERE` clause. We have four predicate groups, each joining their respective predicates using the `AND` operator. Although we cannot further expand the composite clause, we can simplify it and prune unnecessary predicate groups using some rules. Even a rudimentary set of rules can achieve a significant reduction in the complexity of the composite clause. In our case, for every generated composite clause we apply two rules. They are

1. **Remove predicate groups where more than one column name exists**: The reasoning behind this rule is as follows. Since each predicate group is disjunctively combined with the others, it is not possible to determine a more

simplified expression than including all the results. Hence, we remove the predicate group.

This can be demonstrated in the composite clause that we have generated so far, $\boldsymbol{Q}$, which has four parts conjunctively joined. The first part is `l_discount > 0.06 OR l_discount > 0.08 OR l_quantity > 23`. As *l_discount* and *l_quantity* are mutually exclusive columns, an `OR` between the two must include all the rows from both the columns. As a result, this part of the composite clause can be removed safely.

2. **Remove duplicate predicate groups**: Boolean algebra dictates that the expression $\boldsymbol{A} \wedge \boldsymbol{A}$ simply evaluates to $\boldsymbol{A}$. For example, if the composite clause contained `l_quantity > 25 OR l_quantity > 25`, then the second half is redundant and can be safely removed.

Applying the first rule to our example, we can see that all four predicate groups are eliminated — as they all contain different column names. This means that we don't have to further check for the other rule.

In our example, we have thus concluded that the composite query would have no `WHERE` clause. In other words, we must get the results for the entire *lineitem* table to reliably extract the results for both the queries.

### Predicate Covering

Even after the elimination of irrelevant predicate groups, we are often left with some predicates that can be simplified according to their access patterns. This allows us to further reduce the number of predicates present in the composite `WHERE` clause without any change in its outcome.

For example we might have a predicate group such as `l_quantity < 10 OR l_quantity < 15`. We can see that in this case, the two conditions can be replaced by just `l_quantity < 15` without any change in the output. Similarly, if we have `l_tax < 0.03 OR l_tax > 0.01`, this predicate group can be removed altogether because the entire column *l_tax* is needed in the result. In other words, we can generalize this as: If two predicates are combined in such a way that they access the entire column, then the filtering can be removed. These simplifications are performed for every predicate group. The resulting clause is a simplified version that contains all the results for the queries that it is generated from.

The complete algorithm for creating a composite `WHERE` clause, including how simplifications can be performed, is given in Algorithm 1. Once we have a composite `WHERE` clause generated for two queries, we can recreate the selections such that the constituent parts can extract the results from it.

**Algorithm 1:** Generating a compisite WHERE clause

**Data:** Queries $q_1, q_2$
**Result:** Composite where clause $w$

```
1  fun getCompositeWhereClause(q1, q2):
2      w1 ← cnf(whereClause(q1));
3      w2 ← cnf(whereClause(q2));
4      preds_groups ← predicateOR(w1, w2);
       // Simplify the generated clause
5      removeDuplicates(preds_groups);
6      removeDissimilar(preds_groups);
7      removeSingle(preds_groups);
8      w ← empty;
9      foreach predicate group pg in pred_groups do
10         w ← cover(pg).join(or);
11     end
12     w ← w.join(and);
13     return w;
```

### 3.2.4 Generating a Composite Query

So far we have constructed a composite where clause from two queries. Now we must create the complete batched query plan such that its constituent queries' results are completely contained in the batch query's results.

In addition to including the selections from the individual queries, we also need to consider the problem of the missing information required for proper extraction of results. Consider again the example we have been working on. In $Q_1$, we need tuples from the column *l_discount*, but only if the condition `l_discount < 0.06` is satisfied. And from our composite `WHERE` clause, we can reach the conclusion that in order to extract the results for $Q_1$ from the batched results, we need the column *l_discount* to be present in the first place. Thus, we need to add it to the selections of the batched queries. Similarly, we need to also include the column *l_tax*.

This same line of reasoning must also be applied to $Q_2$. Here we find that to retrieve its results, we must include *l_quantity* too, in addition to the columns dictated by $Q_1$.

Finally, the batched query for obtaining the results of $Q_1$ and $Q_2$ would be

```
Q: SELECT l_quantity, l_extendedprice, l_shipdate, l_tax, l_discount
     FROM lineitem
```

This process is illustrated in Algorithm 2. We repeat these steps iteratively for each query in the group. For instance, with a group of three queries, namely $Q_1$, $Q_2$, and $Q_3$, the first two queries would be batched to provide $Q_{12}$ that is batched again with $Q_3$ to get $Q_{123}$. The final obtained query is the composite for all the queries in the group. When queries are aggregated, the regular batching procedure unfortunately can not be directly applied to the queries. We will consider how such queries can be effectively batched in the following section.

**Algorithm 2:** Generating a composite query

**Data:** Queries $q_1, q_2$
**Result:** Composite query $q$

```
1  fun getCompositeQuery(q1, q2):
2      where ← getCompositeWhereClause(q1, q2);
3      selects ← getSelections(q1) + getSelections(q2);
4      selects ← selects+ getColumnNames(where);
5      from ← getFromList(q1);
6      q ← selects + from + where;
7      return q;
```

**Algorithm 3:** Batching a set of queries

**Data:** List of queries
**Result:** List of batched queries

```
1  fun batch(queries):
2      batched ← emptylist;
3      processed ← 0;
4      while processed < len(queries) do
5          q1 ← queries[proc];
6          res ← q1;
7          foreach batched query bq in batched do
8              if getFromList(q1) = getFromList(bq) then
9                  res ← getCompositeQuery(q1, bq);
10                 batched ← batched − bq;
11                 break;
12             end
13         end
14         batched ← batched + res;
15     end
16     return batched
```

### 3.2.5 Batching Aggregations

We have mentioned earlier that there are some additional points that should be taken into consideration when batching queries that contain aggregations.

The issue arises because of incompatibilities between two aggregated queries. Specifically, the outputs of two aggregated queries, in some cases, cannot be derived from a single composite query plan. Consider, for example, the two queries $Q_1$ and $Q_2$ shown below.

$Q_1$:
```
SELECT l_quantity, COUNT(l_quantity)
    FROM lineitem
    WHERE l_discount > 0.06 AND l_quantity > 15
    GROUP BY l_quantity
```

$Q_2$:
```
SELECT l_quantity, l_discount, count(*)
    FROM lineitem
    WHERE l_discount > 0.08 OR (l_quantity > 23 AND l_tax < 0.02)
    GROUP BY l_quantity, l_discount
```

$Q_1$ groups on the column *l_quantity* whereas $Q_2$ groups on the columns *l_quantity* and *l_discount*. If we then generate a composite query using the steps defined in this section, we have to define which grouping to be used: $Q_1$ or $Q_2$. One possible solution is to determine which query is more *general* in terms of its access criteria and then group the composite query based on this. Setting aside the issue of reliably determining where each of the given queries is located on a general-specific grouping spectrum, an additional matter of concern is that for more specific queries, part of the data still needs to be accessed from the database.

Another approach, which we have selected to resolve this problem, is to *de-aggregate* queries before batching. This involves removing the grouping criteria — `GROUP BY` clause — and removing the selections which group columns. This is a simple yet effective approach because once de-aggregated, the queries can be treated as normal, meaning that a composite query can be easily generated without any alteration to its procedure. Applying this approach to the queries $Q_1$ and $Q_2$ requires us to de-aggregate the queries, which would look like

$Q_{1D}$:
```
SELECT l_quantity
    FROM lineitem
    WHERE l_discount > 0.06 AND l_quantity > 15
```

$Q_{2D}$:
```
SELECT l_quantity, l_discount
    FROM lineitem
    WHERE l_discount > 0.08 OR (l_quantity > 23 AND l_tax < 0.02)
```

When we generate a composite query from the de-aggregated queries, it can be used to extract the results of both $Q_1$ and $Q_2$. Now that we have batched the eligible queries, even in the presence of aggregations, our next step as we have seen in the overview, is *substitution* of relevant operations of a query plan to use a suitable materialized view.

## 3.3 Materialized View Substitution

Once we have batched queries from the incoming list, the next task the system undertakes is materialized view substitution. We essentially pick out, from the cached materialized views, ones that are useful in executing our batched query plan and then substitute the relevant parts of the query plan to utilize the materialized view instead of the database.

Determining if and to what extent a query plan can be substituted is called *derivability*, as discussed in Section 2.2. There are four main types of derivabilities: exact, partial, subsuming, and none, all of which our hybrid MQO system supports.

Finding out whether a query is derivable requires storage of materialized views. We will discuss the specifics of how the cache is designed to this end in Section 3.4, but to better visualize the concept of substitution let us assume that the cache is partly occupied with relevant materialized views. Substitution of a query plan, whether it is batched or otherwise, involves two steps:

1. Finding materialized views that are potential candidates for substitution using a coarse filtering criteria.
2. Finding views that are actually substitutable among these candidate materialized views.

Initially, we need to remove views that we are certain will not be substitutable. This includes removing views that access different tables, for example. After this step is completed, we have a list of materialized views that are *potentially substitutable*. We then, in the second step, check if any of the views can indeed be used for substituting the query plan.

The first step is necessary because, even though in our framework checking for materialized view substitution is a fairly cheap operation, we nonetheless need to weed out unsuitable views by matching table signatures. In other words, we extract views that perform scans on the same table as the view, and then use them for substitutions. There are two main reasons for this choice: firstly, it would be uneconomical to determine whether every materialized view could be substituted in a given query plan, and secondly because we know for a fact that for a view that scans on a relation, say *lineitem*, the possibility of it being used for substitution for a query that scans on another relation, say *customer*, is zero.

This process is slightly different in the case that the query plan contains a `JOIN`. For `JOIN`s, we also need materialized views that, in addition to scanning the join of the same tables, scan on any other combination of the tables. This ensures that a subset of relations from the join could be retrieved from the view whereas the rest are obtained from the database. For instance, if we need to substitute a query plan containing a join between the tables $\boldsymbol{A}$ and $\boldsymbol{B}$, then in addition to looking for materialized views that contain a join over $\boldsymbol{A}$ and $\boldsymbol{B}$, we also need to look for materialized views that contain only table $\boldsymbol{A}$ and only table $\boldsymbol{B}$. This type of derivability is called partial derivabililty, as discussed in Section 2.2.

After the substitution, we could have multiple substituted query plans available. From among these, we need to pick one that is most efficient. As we know from our discussion about conventional query optimization in Section 2.1.2, each query plan has an associated cost that is a quantitative measure of what executing that particular plan entails. The calculation of this cost is highly dependent on the system but in general all database systems incorporate some common measures such as the estimated resource utilization of the query, the time taken to access the required data, and so on. This *cost* serves as a good measure to determine which substituted plan to use for the execution of the query. In the rare case that two substituted plans arrive at the same cost estimates, we pick a plan at random.

To illustrate this, consider that we have three materialized views present in the cache: $\boldsymbol{MV}_1$, $\boldsymbol{MV}_2$, and $\boldsymbol{MV}_3$, as shown below. If we need to substitute the query plan for $\boldsymbol{Q}$ from these queries, we have to first find suitable materialized views from the cache. In our case, we can see that our query joins the relations *supplier* and *partsupp*, so the materialized views containing either a join between the relations, or just the *supplier* relation, or just the *partsupp* relation are needed, which in our case is all three cached views.

Now that we have selected suitable materialized views for substitution, we can see from closer inspection of the predicates in the `WHERE` clauses that $\boldsymbol{MV}_2$ cannot be used for substitution. That is, since our query requires `ps_supplycost > 8268.19` and $\boldsymbol{MV}_2$ contains `ps_supplycost > 10000`, substitution is not possible.

$\boldsymbol{Q}$:
```
SELECT ps_partkey, ps_supplycost, s_suppkey
  FROM partsupp JOIN supplier on ps_suppkey = s_suppkey
  WHERE s_acctbal < 522.79 AND ps_supplycost > 8268.19
```

$\boldsymbol{MV}_1$:
```
SELECT ps_partkey, ps_supplycost, s_suppkey
  FROM partsupp JOIN supplier on ps_suppkey = s_suppkey
  WHERE s_acctbal < 600 AND ps_supplycost > 8000
```

$\boldsymbol{MV}_2$:
```
SELECT ps_partkey, ps_supplycost, ps_availqty
  FROM partsupp
  WHERE ps_supplycost > 10000
```

$MV_3$:
```
SELECT s_suppkey, s_acctbal
  FROM supplier
  WHERE s_acctbal < 703.53
```

But this is not the case for $MV_1$ and $MV_3$. For $MV_1$ and $MV_3$, substitution is indeed possible because of matching projections and `WHERE` predicates. The substitutions of $Q$ with $MV_1$ and $MV_3$ are given below. Finally, to select one query plan, we compare the costs of both the substituted plans and pick the one with the lower cost.

$Q_{MV_1}$:
```
SELECT ps_partkey, ps_supplycost, s_suppkey
  FROM MV_1
  WHERE s_acctbal < 522.79 AND ps_supplycost > 8268.19
```

$Q_{MV_3}$:
```
SELECT ps_partkey, ps_supplycost, s_suppkey
  FROM partsupp JOIN MV_3 on ps_suppkey = s_suppkey
  WHERE s_acctbal < 522.79 AND ps_supplycost > 8268.19
```

The algorithm for facilitating materialized view substitutions is presented in Algorithm 4. It is evident from our discussion that the performance of substitution is heavily dependent on the size and performance of the cache. If the cache is too small, then fewer views will be stored, the result of which is a deterioration in performance. In the next section, we will look at how the cache in our hybrid MQO system is designed to overcome such a situation.

**Algorithm 4:** Finding suitable view substitutions

**Data:** Query plan $q$, List of cached materialized views *views*
**Result:** Substituted query plan

```
1  fun getSubstitution(q, views):
2      keys ← getPermutations(getFromList(q));
3      candidates ← empty list;
4      foreach key k in keys do
5          sub_views ← getViewsForKey(k);
6          foreach view sv in sub_views do
7              if canSubstitute(q, sv) then
8                  candidates ← candidates+ getSubstitution(q, sv);
9              end
10         end
11     end
12     plan ← findCheapestPlan(candidates);
13     return plan;
```

## 3.4 Cache

Cache is one of the primary parts of our hybrid MQO system. It defines where materialized views are stored and how they are accessed, i.e., the physical and logical structure of the views. Furthermore, it employs a replacement policy to remove irrelevant data whenever a threshold is reached, which in our case is First In First Out (FIFO).

Materialized views are stored in the main memory. For efficient and quick access, we have implemented a map for retrieving views from the cache. The map contains key-value pairs, with values being the views themselves and the keys being the tables that they access. Whenever a new view is to be added into the cache, the map is checked for existing views with the same key. When such an entry is not found, one is created and the view is added to the map. Otherwise, we append our materialized view in the already existing entry.

Retrieval of materialized views also follows a similar flow. Whenever the systems needs a materialized view, all the corresponding keys are retrieved. For a simple Select-Project query, there is a singular key whereas in the case of Select-Project-Join queries, there might be multiple keys as explained in Section 3.2.5.

The views are stored and accessed using the mechanisms described above. When the cache reaches a threshold, 80% of the total size in our case, the clean routine is called which removes irrelevant views from the cache to make space for newer and more relevant ones. We have employed a conventional Least Recently Used (LRU) policy, where the views that are outdated are removed. This policy runs until 10% of the cache size is reclaimed.

We have so far looked at the individual components that make up the hybrid MQO system: the batcher, the substituter, and the cache. In the following section, we will understand how these three components jointly form our hybrid MQO system. We will also visualize the process of batching and substitution through relevant examples.

## 3.5 Integration

Having discussed in detail about the discrete components of our hybrid MQO system, we will now turn to considering them as a single entity. Given a set of queries, we will see how the three individual parts of the hybrid MQO system, namely the batcher, the substituter, and the cache work together to efficiently and accurately execute multiple queries in a shared fashion.

As we can see in Figure 3.1, the given list of queries is initially parsed and batched together. From our study of the Batcher, we know that this involves finding out

eligible queries from this list and then generating composite queries whose results completely envelope the results of its parts.

These generated composite queries go through a phase of logical optimization where the parsed representation of a query — a tree in this instance — is converted to a logical plan. Each logical plan then goes through the substituter which is responsible for using materialized views stored in the cache to aid in the execution of the composite query. If suitable views are found in the cache, relevant parts from the logical plan are replaced with their equivalent materialized view configurations. In case this is not possible, the logical plan moves to the next step unchanged.

In the next step, the logical plan — which may be substituted or otherwise — goes through physical optimization. Depending on factors such as the ones described in Section 2.1.2, a physical plan is generated that specifies implementational details about the various operators present. This physical plan is then executed and the results are presented to the user.

Consider that the system receives as input, the list of queries presented below. We also assume, for the purposes of this illustration, that the cache is currently empty. This example illustrates the versatility of the hybrid MQO method quite well because the queries are of different types: Select-Project, Select-Project-Join, and Select-Project-Join-Aggregate.

$Q_1$:
```
SELECT s_name, s_suppkey, s_acctbal, c_custkey
    FROM supplier JOIN customer on s_nationkey = c_nationkey
    WHERE s_suppkey < 6870 OR s_acctbal < 145.72
```

$Q_2$:
```
SELECT s_name, n_nationkey
    FROM supplier JOIN nation on s_nationkey = n_nationkey
    WHERE n_regionkey = 1 AND s_suppkey < 5000
```

$Q_3$:
```
SELECT s_name, sum(s_suppkey)
    FROM supplier
    WHERE s_acctbal < 100
    GROUP BY s_name
```

$Q_4$:
```
SELECT l_quantity, AVG(l_tax)
    FROM lineitem
    WHERE l_quantity < 25 AND l_discount < 0.03
    GROUP BY l_quantity
```

$Q_5$:
```
SELECT l_discount, l_tax
    FROM lineitem
    WHERE l_discount < 0.06 AND l_extendedprice < 10000
```

$Q_6$:
```
SELECT l_quantity, SUM(l_extendedprice)
    FROM lineitem join partsupp ON l_partkey = ps_partkey
    WHERE ps_supplycost < 500 AND l_tax < 0.05
```

Figure 3.2 shows how the given set of six queries can be executed in our hybrid MQO system. We will refer to it along the way during the explanation of query execution.

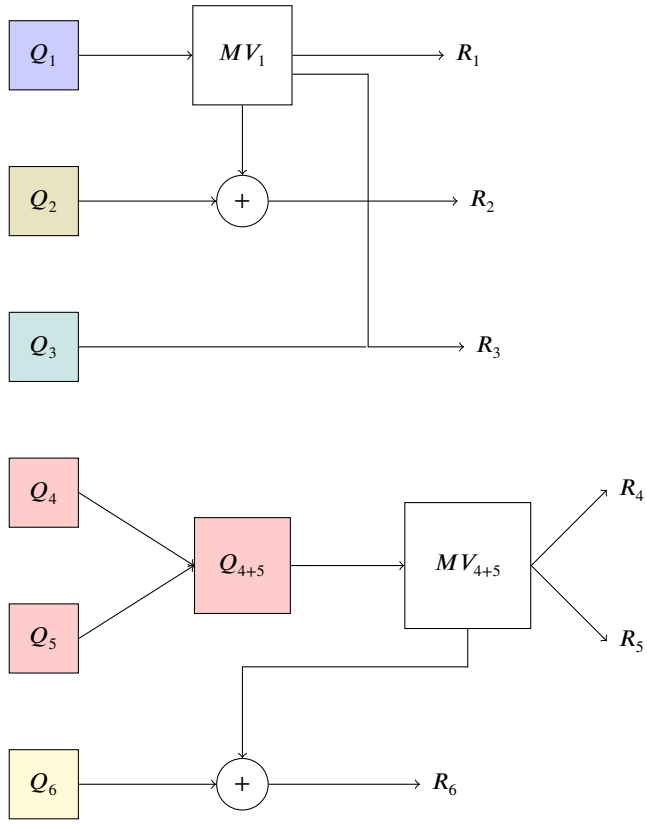



Figure 3.2: Functional example of our hybrid MQO method

Firstly, the batcher segregates those queries that can be batched together and those that cannot. Since our batching eligibility criteria is that the queries be accessing the same tables, we can say that $Q_4$ and $Q_5$ can be batched. But the remaining queries, unfortunately, cannot. Although batching is not possible for some queries, we still have the possibility of achieving performance gains through materialized view substitution.

In the next step, the optimizer checks for any existing materialized views that can be used to execute $Q_1$. Since our cache is empty at this point, there is no possibility of finding any such views. In that case, a materialized view representing $Q_1$ is generated, labelled in Figure 3.2 as $MV_1$. This allows us to get the result for $Q_1$ from $MV_1$, and also allows us to cache $MV_1$ for future use.

When the system encounters $Q_2$ and checks the cache to find any suitable views, even though the view that we have stored, $MV_1$, does not directly match the entire query, we can see that one of the relations from the `JOIN` of $Q_2$, *supplier*, can be derived

from the materialized view. So the relation *nation* is obtained from the database and *supplier* from the view. In this way, $Q_2$ is optimized.

For $Q_3$, a similar approach is applicable. Studying the `WHERE` clause of $Q_3$, we can see that its results can also be obtained from the materialized view. Additionally, unlike in the case of $Q_2$, we do not have to retrieve any additional information from the database. All the required tuples to get the result for $Q_3$ are present in $MV_1$.

Next, the optimizer processes $Q_4$ and $Q_5$ in tandem. Since these queries can be batched together, the batcher creates a composite query as explained in Section 3.2. For this composite query, we check if any existing materialized view can be substituted. In our case, the only materialized view present is $MV_1$, which does not scan on the relation *lineitem*, so we can directly conclude that $MV_1$ is not applicable. Having found no possible candidates for substitution, we create a materialized view from the composite query, $MV_{4+5}$, as shown in the figure. Because we know that our composite query contains all the tuples from both $Q_4$ and $Q_5$, we can safely use $MV_{4+5}$ for substitution. In other words, we can derive the results for $Q_4$ and $Q_5$ from $MV_{4+5}$.

Finally, to execute $Q_6$, the optimizer goes through similar steps as for the previous queries. We check whether any materialized view can be used to derive the results and conclude that $MV_{4+5}$ can indeed be used to derive a partial result. $Q_6$ is a join of the relations *lineitem* and *partsupp* and the information about *lineitem* can be obtained from the materialized view. The tuples for the relation *partsupp* are fetched from the database and then the join is performed.

This is the process by which our hybrid MQO method executes queries in a shared context efficiently. In the following section, we will discuss briefly discuss its implementational details.

## 3.6 Realization of Hybrid MQO

Thus far we have looked, in detail, at the workings of our hybrid MQO approach. We have studied creation of composite plans, materialized view substitutions, and the design of the cache. In this chapter, we will briefly discuss the specifics of implementing this approach. We initially start with a discussion about Apache Calcite [BRH+18], which is the framework used for the core of the optimization. Then we look at how Calcite can be used to facilitate materialized view substitutions. Finally, we look at how this can be combined with batching queries within the context of Apache Calcite to form the complete hybrid MQO system. The complete code for the implementation of our hybrid MQO method is available on Github [1].

[1] `https://github.com/vasudevrb/mqo`

### 3.6.1 Apache Calcite

Apache Calcite [BRH+18] is a framework, written in Java, that enables query optimization support to popular database systems. Even though Calcite provides a lot of the functionality of a DBMS, it is not actually a complete database management system. For instance, unlike a DBMS, it does not dictate how the data is precisely stored in a physical sense. Rather, it is only concerned with processing the data. The two main principles it is built on are modularity and extensibility. The complete framework is modular; if some particular functionality is not required, it is possible to fallback to the database defaults. Owing to its modularity, it is possible in Calcite to process data from heterogeneous data sources simultaneously. Extensibility in Calcite allows the addition (or redesign) of its existing components using a simple and intuitive interface.

Although Calcite allows for many possible operations related to databases such as query validations and heterogeneity in data-sourcing, it is primarily designed for query optimization. The optimizer in Calcite works by applying rules to a logical query plan. There are hundreds of default rules available but custom ones can also be written since Calcite is extensible. Calcite provides two planner engines: Heuristic planner where pre-defined rules are applied regardless of their impact on the cost of the plan, and Volcano planner where cost-based optimization of the query plan is performed. We have discussed rule-based and cost-based optimizations in more detail in Section 2.1.2.

### 3.6.2 Materialized View Substitution

One of the prominent features of Calcite is that it provides an interface for substituting query plans by using materialized views, based on the approaches suggested by [GL01, CKPS95].

Calcite has its own concept of a materialized view, which some databases such as Cassandra have incorporated into their architecture [BRH+18]. Materialized view substitution in Apache Calcite works as follows: the query plan and the materialized view are registered in the planner along with unification rules that determine the extent of substitution possible. The advantage of unification rules is that partial derivability, in addition to exact and subsuming derivabilities, is also feasible.

### 3.6.3 Batching

Because the implementation of Apache Calcite is in Java, to preserve compatibility we have also implemented out batching algorithm and the cache in the same environment. The implementation of the batcher follows the structure laid out in Section 3.2.

Throughout this chapter, we have concerned ourselves with the hybrid MQO technique. Beginning with an overview of the complete system, we then turned to look at each

of the specific parts in detail before discussing how these parts form a whole. In the following chapter, we will evaluate the performance of the system and assess its benefits and limitations.

# 4 Evaluation

We measure the performance of our hybrid Multi-Query Optimization (MQO) method using queries of various types, sizes, and derivabilities. We use such an approach as our method is susceptible to the complexity of a query. For example, joins and aggregates are more complex and take longer to execute than filters. The performance also depends on the size of the query result: queries with larger results tend to take longer to execute and so caching these results would, if the queryload is similar, benefit more from our hybrid MQO technique.

One of the most important factors of optimized execution is the *derivability* of the queryload. Here, derivability is the quantitative measure for the amount of similarity a set of queries contain. For instance, if we have a set of 100 queries, a derivability of 50% is where half of query results can be derived directly from results of the other half. However, this metric relies heavily on the order that the queries are presented in. For example, if $\boldsymbol{Q}_2$ is derivable from $\boldsymbol{Q}_1$, it does not necessarily imply the inverse. As we can infer from Section 2.2 and Figure 2.5, only exact derivability is commutative in nature. Further information about the importance query order is given in Appendix A.

Moreover, if a set of queries has lower derivability, then there would be fewer chances for materialized view substitution, possibly leading to poor optimization. For highly derivable queries, the performance improves because of a higher number of materialized view substitutions. We have evaluated our hybrid MQO technique on a range of different derivabilities to accurately establish its impact on optimization.

This chapter is organized as follows. In Section 4.1 we present a detailed account of our test setup. This includes Section 4.1.1, which explains our choice of dataset. Then, in Section 4.1.2, we describe the query generation process at length, outlining the different types of queries used to generate diverse query workloads. Following this, we give a brief overview of the test environment in Section 4.1.3. Thereafter, we study the performance of the hybrid MQO system for many different query configurations, in Section 4.2. Finally, with Section 4.3, we examine the results more closely and present reasons for the observed trends in performance.

## 4.1 Evaluation Setup

For testing our hybrid MQO system, we focus on simulating real-world query workloads including queries with all supported operators. To this end, we create query workloads from the TPC-H dataset [1]. The relevance of TPC-H dataset in generating queryloads is explained in Section 4.1.1. The actual generation of queryloads is detailed in Section 4.1.2. Subsequently, we performed all of our testing on PostgreSQL installed on a virtual machine, the specifics of which are presented in Section 4.1.3.

### 4.1.1 Dataset

For evaluation of our hybrid MQO technique, we use the TPC-H dataset with a scale factor of SF-5. In addition to being a standardized benchmark for database applications, TPC-H has certain properties that can be used to test different aspects of our hybrid MQO method. Even though we generate custom queryloads to test our hybrid method, looking at the benchmark queries can help us determine certain characteristics of the underlying data. Figure 4.1 shows the frequency of each TPC-H table in the benchmark queries.

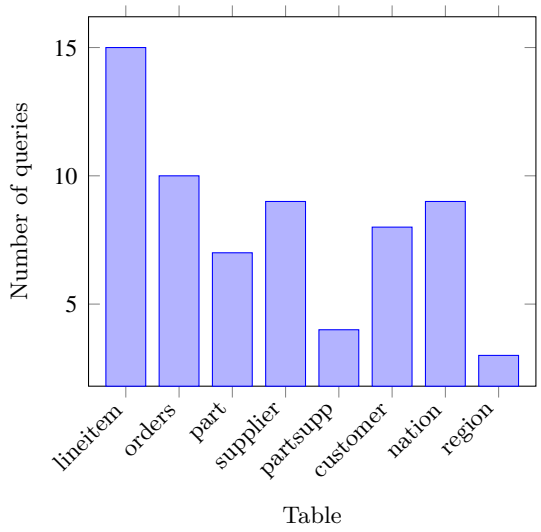



Figure 4.1: TPC-H benchmark query composition

We can observe from Figure 4.1 that the tables *lineitem* and *orders*, as well as being the largest by size (*lineitem*: $\boldsymbol{SF} * 6,000,000$ rows, *orders*: $\boldsymbol{SF} * 1,500,000$ rows), are also the frequently accessed ones (*lineitem*: 15 queries, *orders*: 10 queries) in the benchmark queries. This shows that over the entire range of queries there is some degree of overlap. Similarly, *part*, *supplier*, *nation*, and *customer* are moderately used

[1] `https://www.tpc.org/tpch/`

among the queries, whereas *partsupp* and *region* are infrequent. The *region* table not having a significant overlap in the queries is inconsequential because it is the smallest by size, containing only 5 rows.

This analysis demonstrates the viability of using TPC-H for evaluating our hybrid MQO method. Unfortunately, the benchmark queries cannot be used in their present form, without any modifications, because they do not cover all of our test cases. For example, one of our main aims, which is to determine the impact of query similarities on the performance of our hybrid method, would be infeasible with only 22 queries. For this reason, we generate custom queryloads — which are inspired by the benchmark queries — for the dataset, the details of which are presented in the following section.

### 4.1.2 Query Generation

As we have discussed in the previous chapters, certain queries have an inherent advantage when considering multiple query optimization techniques. Usually such queries are analytical in nature. For generating appropriate queryloads we focused on the three factors explained in the introduction to this chapter: the type of the query, the result size of the query, and the derivability in the context of the complete set of queries.

For the queryload, we generate 320 different queries based on 32 templates by varying their predicate values. These 32 template queries, which can be grouped into 5 types, are created from a combination of different operators. The 5 types of queries and their respective quantities are shown in Table 4.1. Furthermore, all template queries are presented in Appendix A.

| Type of queries | Quantity |
|---|---:|
| Single column filtering | 9 |
| Multiple column filtering | 4 |
| Join | 5 |
| Aggregate | 7 |
| Join-Aggregate | 7 |
| Total | 32 |

Table 4.1: Composition of query templates

By careful selection of initial predicate values of every template and by using deterministic random numbers, we can vary the result size, and thereby the derivability of the entire queryload. Additionally, by including these different types of queries we can ensure that each queryload contains enough variability to fully use the characteristics of our hybrid MQO system. The result size of these queries varies from a few bytes

to hundreds of MB. An example of a template query is shown below, with values denoting the range for respective predicates, which are substituted during runtime. Here, the value of `o_totalprice` is randomly picked between 0 and 60000, and that of `c_acctbal` is between -1000 and 5000.

```
// [0 60000] [-1000 5000]
SELECT c_name, avg(c_acctbal), avg(o_totalprice)
FROM orders JOIN customer on o_custkey = c_custkey
WHERE o_totalprice < %.2f AND c_acctbal < %.2f
GROUP BY c_name
```

Using this approach, we can generate queryloads with differing derivabilities. Each set of 320 queries corresponds to a derivability ranging from 0% to 90%.

Additionally, the type of derivability that a query is also affects the results we observe. In Figure 4.2, we plot the split-up of diifferent derivability types in a generated queryload. As we expect, for a 0% derivable queryload none of the queries are derivable. The proportion of non-derivable queries decreases as the derivability increases. For a moderately derivable queryload of 60%, slightly more than half of the queries are derivable. Among these, most are subsumed by materialized views. At the maximum derivability of 90%, a substantial number of queries are derivable, with more than half being subsumed, and a little over a quarter being exactly derivable. For all queryloads, the proportion of partially derivable queries is smaller than subsuming derivability because in our method, partial derivability is only applicable for queries containing joins. In such case, if some table present in the join can be read from a materialized view, it is considered as partially derivable.

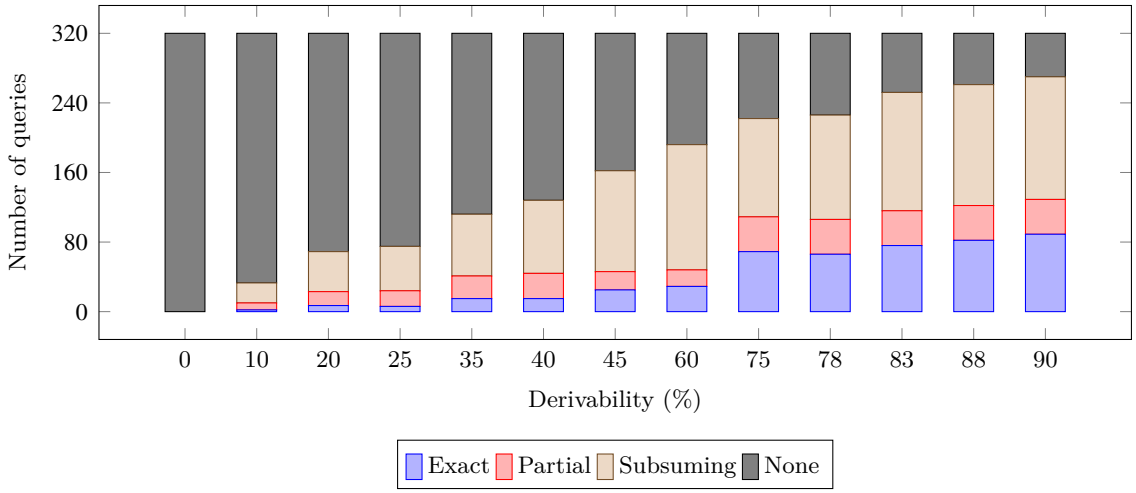



Figure 4.2: Proportion of different types of derivabilities in generated queryloads

Queries belonging to different derivability types provide different reasons for the trends observed in the execution summary. For instance, the prevalence of exactly derivable

queries in a queryload implies that a marginal improvement in performance of our hybrid method is not particularly noteworthy. As exact derivability requires precisely the tuples present in a materialized view, it is the most efficient to compute, making an improvement of, say 2x over a queryload of 320 queries, substandard. Similarly, partial and subsuming derivabilities also carry some underlying assumptions about the queries that help explain the overall performance. We take a look at the effect of derivability types on our observed result in Section 4.3.

### 4.1.3 Test Environment

All our testing was done on a virtual machine on Google Cloud. Specifically, the E2-Highmem instance with a storage of 100GB and main memory of 32GB was used. We used PostgreSQL as our main database, imported with data from the TPC-H benchmark with a scale factor of SF-5.

We have tested our generated queryloads with four different execution strategies.

1. **Sequential execution**, indicating conventional query execution where each query is executed in isolation, without any shared optimizations.

2. **Shared Sub-Expressions (SSE)** with batch sizes of 4-6, where shared query plans are generated and executed. The results of the constituent queries of the batch are extracted from the total result.

3. **Materialized View Reuse (MVR)**, where materialized views are stored and reused for optimization, but batching of queries is not performed.

4. **Hybrid execution** with batch sizes of 4-6, where the optimizations as detailed in Chapter 3 are performed.

## 4.2 Performance Analysis of the Hybrid MQO

In Section 4.1 we have detailed the process of generating suitable queryloads and the different execution strategies. In this section, we will present our findings in three parts, each part focusing on the impact of a certain trait on the performance of our hybrid MQO system. In Section 4.2.1 we study how different derivabilities and cache sizes influence the execution times of different approaches. Following this, in Section 4.2.2 we compare the performance gains and losses of different execution strategies over the whole range of tested queryloads. Finally, Section 4.2.3 studies the impact of different relational operators on the performance.

### 4.2.1 The Effect of Derivability and Cache Size

The time taken to execute different queryloads under different execution strategies with varying cache sizes is depicted in Figure 4.3. In the figure, the horizontal axis shows the respective cache size in MB, and the vertical axis indicates the execution time of the configuration.

Firstly, we can see that, as sequential and Shared Sub-Expressions (SSE) executions have no effect from caching, the time taken to execute queries using these techniques is independent of the cache size, as expected. However, since the other two approaches — Materialized View Reuse (MVR) and hybrid execution — reuse materialized views, their execution times change relative to the size of the cache.

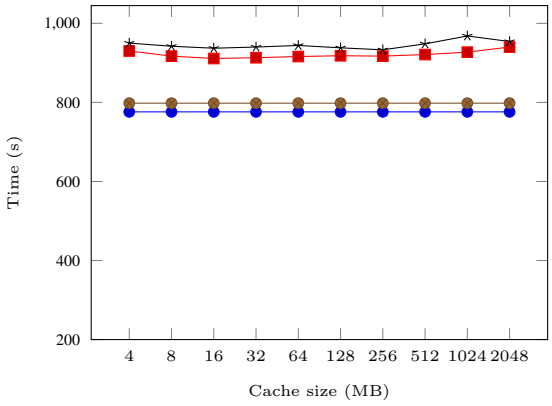
For lower derivabilities, i.e. 0% and 25%, the performance of MVR and hybrid execution is poorer than that of sequential and SSE. In the case of 25% derivability however, we can observe that there is a marginal improvement in the execution time of MVR and hybrid execution when the cache is large. The reason behind this poor performance for lower derivabilities and smaller caches is that since fewer queries are cached, the time spent on creating, caching, and probing materialized views appends to the normal execution time.

As the derivability increases, the benefits of reusing materialized views become clear. As we can see in Figure 4.3c, for 50% derivability, as the cache size exceeds 256MB, there is a drastic reduction in the execution times for MVR and hybrid executions. This cache size threshold depends on the result size of the queryload. For our queryload, it is at this point that an adequate amount of views are stored that can be used in aiding subsequent executions. This pattern of results holds for all higher derivabilities.

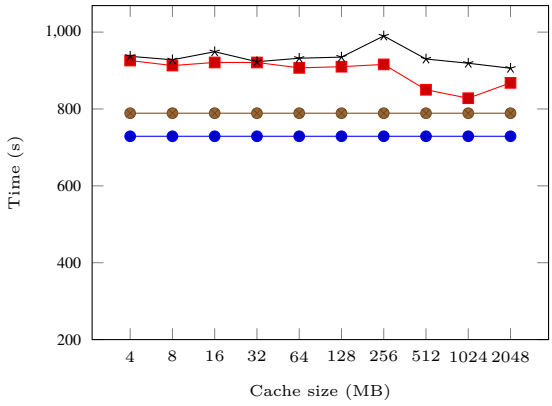
As the derivability further increases beyond 50%, we see the performance disparity between MVR and the hybrid method widening for smaller caches. Previously, MVR and hybrid method yielded similar times. But when the derivabilities are higher and the cache is small, the hybrid method outperforms MVR by almost a factor of 2. This effect is not observed for larger caches. At 90% derivability, the hybrid method executes queries faster as the cache gets bigger, reaching almost twice the speed of sequential and SSE executions at largest caches.

Overall, for low derivabilities the hybrid method generally underperforms when compared to the other three approaches. As the derivability increases, though, the execution times decrease with the bulk of reduction being observed when the cache is sized at 256MB.

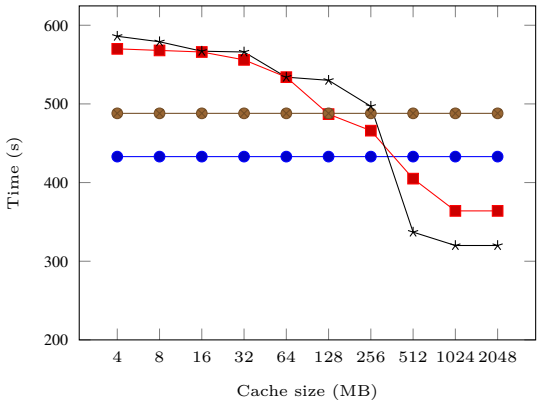
Furthermore, when the derivability is high, the performance gains of the hybrid method over MVR are quite pronounced for smaller caches. And for larger caches, the effect is reversed: the performance gains over SSE, as opposed to MVR, are substantial.



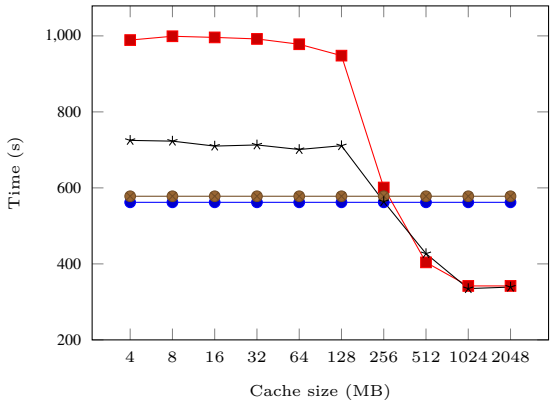

(a) 0% Derivability



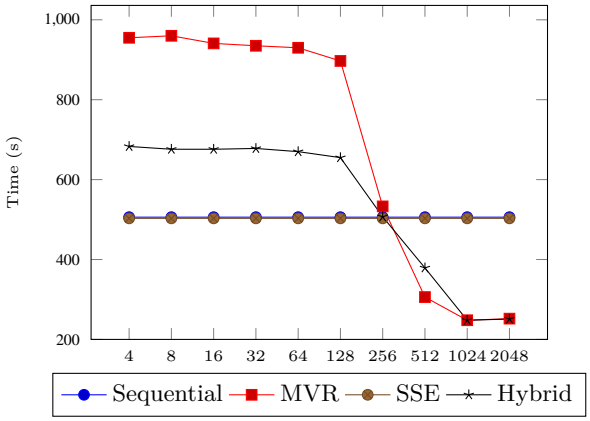

(b) 25% Derivability




(c) 50% Derivability




(d) 75% Derivability




(e) 90% Derivability

Figure 4.3: Execution times of different execution strategies

In this section, we have seen how execution times are influenced by varying derivabilities and cache sizes. Since different queryloads contain fundamentally different queries, an absolute comparison of the execution times for different derivabilities is ineffective. For instance, a 40% derivable queryload may execute twice as fast as a 90% derivable queryload. This does not represent a degradation in performance because the queries in each are different. In order to compare the performance of our hybrid system across different derivabilities, we instead use a relative performance measure. This is presented in the following section.

### 4.2.2 Performance Comparison of Execution Strategies

Figure 4.4 depicts heatmaps of performance gains and losses of our hybrid MQO method as compared to sequential, SSE, and MVR respectively. The number in each cell indicates the relative performance variation of the corresponding hybrid MQO execution. For instance, a value of 0.778 (*Derivability*: 75%, *Cache size*: 8MB) in Figure 4.4a indicates that our hybrid method is 30% slower in executing the queryload. Similarly, a value of 2.017 (*Derivability*: 90%, *Cache size*: 2048MB) represents that hybrid MQO completed the execution twice as fast as sequential. Moreover, for easier visualization of global trends in performance, the cells are colored, with green representing better hybrid execution performance and red representing worse.

In Figure 4.4a we compare our hybrid method with sequential execution. Consistent with our assumptions, we can see that the lower right section of the figure is highlighted green, signifying better performance than sequential execution for higher derivabilities and larger caches.

Observing the heatmap in more detail, we see that for lower derivabilities, our hybrid method is approximately 25% slower regardless of cache size. As the derivability increases, the speedup factor becomes greater than 1, with 50% derivable queryload exhibiting a 25%-30% gain in performance. Further increases in the derivability and the cache size produce better performance still, with 90% derivable queryload being twice as fast as sequential execution.

In some cases, we can observe that having a larger cache harms the performance of our hybrid system. This counter-intuitive phenomenon can be clearly observed in the case of 35% derivability with cache sizes 128 and 256MB. The former configuration is 25% slower but the latter is 30% slower. Upon closer inspection, this issue seems unique to Apache Calcite and its handling of materialized view substitutions. Specifically, when the stored materialized view is very large and the derivable query has no rows in its result, Calcite does not use data statistics to infer that the query yields nothing as conventional query processors usually do, relying instead on extracting the results from the materialized view. Since cycling through the entire materialized view is expensive, this disparity is observed.

| Derivability (%) | Cache Size (MB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| 0 | 0.817 | 0.824 | 0.829 | 0.826 | 0.822 | 0.827 | 0.832 | 0.819 | 0.802 | 0.814 |
| 10 | 0.843 | 0.854 | 0.828 | 0.840 | 0.839 | 0.860 | 0.798 | 0.862 | 0.844 | 0.877 |
| 20 | 0.829 | 0.820 | 0.829 | 0.837 | 0.832 | 0.844 | 0.781 | 0.838 | 0.851 | 0.867 |
| 25 | 0.779 | 0.786 | 0.768 | 0.790 | 0.783 | 0.780 | 0.737 | 0.784 | 0.794 | 0.805 |
| 35 | 0.735 | 0.751 | 0.758 | 0.762 | 0.749 | 0.764 | 0.708 | 0.760 | 0.766 | 0.790 |
| 40 | 0.742 | 0.747 | 0.743 | 0.763 | 0.769 | 0.814 | 0.837 | 0.770 | 0.931 | 0.942 |
| 50 | 0.739 | 0.747 | 0.764 | 0.766 | 0.812 | 0.817 | 0.872 | 1.285 | 1.351 | 1.352 |
| 60 | 0.723 | 0.728 | 0.752 | 0.747 | 0.745 | 0.749 | 0.909 | 1.201 | 1.636 | 1.629 |
| 75 | 0.776 | 0.778 | 0.792 | 0.788 | 0.802 | 0.790 | 0.997 | 1.318 | 1.678 | 1.660 |
| 78 | 0.760 | 0.758 | 0.767 | 0.740 | 0.763 | 0.762 | 0.934 | 1.346 | 1.825 | 1.837 |
| 83 | 0.768 | 0.768 | 0.774 | 0.774 | 0.772 | 0.779 | 0.996 | 1.731 | 2.013 | 1.991 |
| 88 | 0.808 | 0.817 | 0.812 | 0.816 | 0.813 | 0.810 | 0.955 | 1.535 | 2.530 | 2.545 |
| 90 | 0.740 | 0.748 | 0.748 | 0.746 | 0.755 | 0.772 | 0.998 | 1.336 | 2.037 | 2.017 |

(a) Hybrid vs. sequential execution

| Derivability (%) | Cache Size (MB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| 0 | 0.840 | 0.848 | 0.852 | 0.849 | 0.845 | 0.851 | 0.856 | 0.842 | 0.824 | 0.837 |
| 10 | 0.871 | 0.882 | 0.856 | 0.868 | 0.866 | 0.888 | 0.824 | 0.891 | 0.872 | 0.906 |
| 20 | 0.867 | 0.858 | 0.867 | 0.876 | 0.870 | 0.883 | 0.817 | 0.876 | 0.889 | 0.907 |
| 25 | 0.843 | 0.851 | 0.832 | 0.855 | 0.847 | 0.844 | 0.797 | 0.849 | 0.859 | 0.871 |
| 35 | 0.824 | 0.842 | 0.850 | 0.855 | 0.841 | 0.857 | 0.794 | 0.853 | 0.860 | 0.886 |
| 40 | 0.874 | 0.879 | 0.875 | 0.898 | 0.905 | 0.958 | 0.985 | 0.906 | 1.096 | 1.108 |
| 50 | 0.832 | 0.842 | 0.861 | 0.862 | 0.914 | 0.920 | 0.982 | 1.448 | 1.522 | 1.523 |
| 60 | 0.761 | 0.767 | 0.793 | 0.787 | 0.785 | 0.789 | 0.958 | 1.266 | 1.724 | 1.717 |
| 75 | 0.798 | 0.800 | 0.815 | 0.811 | 0.824 | 0.813 | 1.026 | 1.355 | 1.725 | 1.707 |
| 78 | 0.770 | 0.769 | 0.778 | 0.750 | 0.773 | 0.773 | 0.947 | 1.365 | 1.851 | 1.863 |
| 83 | 0.762 | 0.762 | 0.767 | 0.767 | 0.765 | 0.772 | 0.988 | 1.716 | 1.996 | 1.974 |
| 88 | 0.788 | 0.798 | 0.793 | 0.797 | 0.794 | 0.791 | 0.932 | 1.498 | 2.470 | 2.485 |
| 90 | 0.736 | 0.744 | 0.743 | 0.741 | 0.751 | 0.768 | 0.992 | 1.328 | 2.025 | 2.005 |

(b) Hybrid vs. SSE execution

| Derivability (%) | Cache Size (MB) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 |
| 0 | 0.978 | 0.974 | 0.973 | 0.971 | 0.971 | 0.979 | 0.983 | 0.971 | 0.957 | 0.986 |
| 10 | 0.974 | 0.985 | 0.960 | 0.984 | 0.972 | 1.003 | 0.956 | 1.003 | 0.954 | 1.018 |
| 20 | 0.993 | 0.976 | 0.989 | 0.996 | 0.966 | 0.988 | 0.936 | 0.941 | 0.908 | 0.975 |
| 25 | 0.988 | 0.984 | 0.970 | 0.998 | 0.974 | 0.973 | 0.925 | 0.914 | 0.901 | 0.958 |
| 35 | 0.946 | 0.956 | 0.967 | 1.016 | 0.998 | 1.006 | 0.860 | 0.885 | 0.872 | 0.910 |
| 40 | 1.032 | 1.064 | 1.033 | 1.040 | 1.045 | 1.107 | 1.103 | 0.944 | 1.075 | 1.083 |
| 50 | 0.972 | 0.980 | 0.999 | 0.982 | 1.001 | 0.918 | 0.939 | 1.201 | 1.135 | 1.136 |
| 60 | 1.248 | 1.247 | 1.285 | 1.258 | 1.252 | 1.226 | 1.117 | 1.165 | 1.387 | 1.270 |
| 75 | 1.365 | 1.383 | 1.404 | 1.391 | 1.394 | 1.332 | 1.066 | 0.947 | 1.019 | 1.009 |
| 78 | 1.425 | 1.431 | 1.430 | 1.382 | 1.428 | 1.372 | 1.012 | 0.989 | 1.037 | 1.048 |
| 83 | 1.431 | 1.431 | 1.404 | 1.413 | 1.385 | 1.365 | 1.021 | 1.298 | 1.023 | 1.030 |
| 88 | 1.333 | 1.367 | 1.315 | 1.330 | 1.322 | 1.328 | 1.018 | 1.097 | 0.980 | 1.011 |
| 90 | 1.399 | 1.419 | 1.392 | 1.379 | 1.389 | 1.370 | 1.052 | 0.808 | 0.999 | 1.007 |

(c) Hybrid vs. MVR execution

Figure 4.4: Relative performances of execution strategies compared to the hybrid method

Figure 4.4b shows the relative performance of our hybrid method compared to SSE, which is similar to Figure 4.4a. Here too, our hybrid method underperforms for lower derivabilities and smaller caches, but as these increase there is an improvement in the performance.

Figure 4.4c shows a comparison of the relative performance of our hybrid method with MVR execution across all derivabilities and cache sizes. The results here reflect our observation from Section 4.2.1 that the performance gains when comparing hybrid and MVR are more substantial for smaller caches. For highly derivable queryloads with large caches, most performance values lie between 1 and 1.2, indicating only a minor improvement.

Furthermore, we can observe a particular detail about the relationship of hybrid MQO method with its constituent parts: SSE and MVR, by comparing Figure 4.4b and Figure 4.4c. For queryloads with high derivability, the performance of our hybrid method exceeds the baseline of the basic MVR approach (Figure 4.4c) when the cache is small. This observation suggests that sharing sub-expressions contributes towards

this increase in performance. Similarly, for larger caches, the hybrid method is much more efficient when compared to SSE, as shown in Figure 4.4b, implying that the efficiency is a result of materialized view reuse.

In this section, we have compared our hybrid MQO method to three other execution strategies. Overall, we see a 2x speed-up over sequential and SSE when the queries are derivable and the cache is large. So far, we have only looked at the complete queryload to understand the strengths and weaknesses of our hybrid method. However, examining the role that different query operators play in forming the overall performance metric is just as important.

### 4.2.3 Impact of Query Types

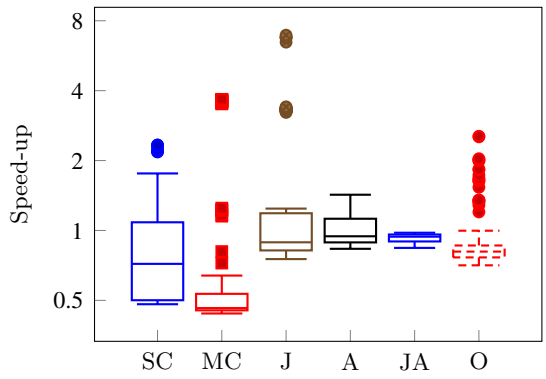
As we have seen in Section 4.1.2, every queryload of 320 queries is generated from 32 template queries, which can be divided into 5 types based on their operators, as shown in Table 4.1. In this section we compare the performance of our hybrid method to the other three executions for queries belonging to a similar type. The speedup of the hybrid method over sequential, SSE, and MVR executions is shown in Figure 4.5.

Taking an overview of the individual plots, we can notice that the performance of our hybrid method varies considerably depending on the type of queries. The huge variation observed in the plots is representative of the performance for the entire range of derivabilities. Low derivabilities severely diminish the performance of our hybird method and high derivabilities severely enhance. Figure 4.5a shows the variability in terms of the hybrid method's speed-up over sequential execution.

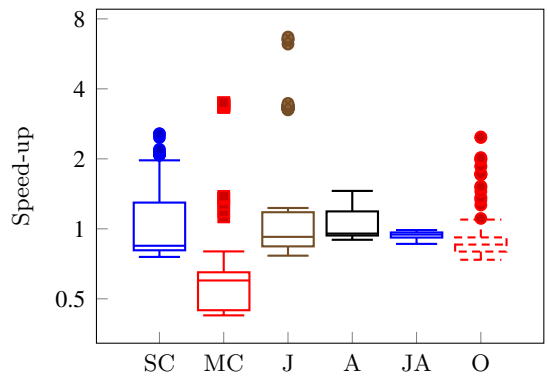
For filter queries, we observe large fluctuations between the extremes, with minimum and maximum speed-ups of around 0.5 and 2 respectively. Closer inspection of the execution data reveals that, as we have observed before, higher derivabilities lead to gains in performance and lower derivabilities lead to losses. Multiple column filter queries, on average, performed poorer than single column filter queries, with most multi-column filter queryloads being slower to execute with our hybrid method than with sequential.

However, operators with a higher degree of complexity, such as joins and aggregates, demonstrate an improved performance over filter queries. This further supports the observation that MQO is more beneficial for heavier operators than lighter ones, as we have seen in Section 2.2. For high derivabilities, queryloads containing exclusively joins show a speed-up of over 6x.

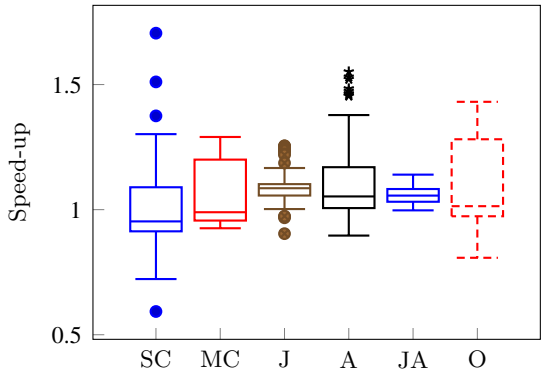
Although queryloads containing exclusively joins *or* aggregates show more variability, they are also, on average, more efficient than queryloads containing joins *and* aggregates.




(a) Hybrid vs. sequential execution




(b) Hybrid vs. SSE execution




**SC**: Single column filter
**MC**: Multi column filter
**J**: Join
**A**: Aggregate
**JA**: Join-agggregate
**O**: Overall

(c) Hybrid vs. MVR execution

Figure 4.5: Relative performances of individual query types

As a whole, mixed queryloads with filters, joins, and aggregates show a speed-up that is insignificant. Because our hybrid method performs poorly for queries that are not derivable, plotting its performance over the entire range of derivability tends to offset the gains observed for highly derivable queries.

Figure 4.5b shows the speed-up of our hybrid method over SSE, which bears a lot of similarity to Figure 4.5a. Here also, filter queries show less of a speed-up compared to join and aggregate queries, with the overall speed-up equivalent to that of Figure 4.5a.

On the other hand, Figure 4.5c, which plots the relative speed-up of our hybrid method over MVR, shows a different pattern. When comparing the performance of filter queries, the hybrid method shows a maximum speed-up of 1.7x. As we have seen in the previous section, most of the gains of our hybrid method over MVR exist for smaller caches. The performance of the hybrid method is equivalent to that of MVR when comparing larger caches.

The performance for joins and aggregates varies from a slow-down of 0.9x to a speed-up of 1.5x, with joins showing a lower variability as well as tighter bounds. Join-aggregate queryloads maintain a similar form here as in the other two plots, showing only marginal improvements on average. Overall the median performance speed-up is around 1.1x, indicating that even with the additional resources spent on creating, caching, and probing materialized views, the execution does not suffer greatly.

Altogether, under high derivable workloads, our hybrid method performs far better when compared to sequential (2x) and SSE (2x) executions than MVR (1.4x). This mirrors our conclusions from previous figures.

## 4.3 Discussion

The previous section presented our results comparing our hybrid MQO method to three other execution strategies. Based on our findings, the general trend implies that for good performance of the hybrid system, a large cache as well as queries with high derivability are essential. This remains the case whether queryloads are homogeneous in their operator composition, or heterogeneous.

The derivability of a queryload specifies the amount of queries that can extract their results from other query results. Derivability is highly sensitive to the order of queries, as the earlier occurring query must be wider in scope than the latter one. The derivability types that constitute each queryload are depicted in Figure 4.2. As the number of exactly derivable queries is not in the majority, we can conclude that the observed speedup is due to the optimizations of our hybrid method, as opposed to the relative computational efficiency of deriving exact results from materialized views. Furthermore, we also observe that the performance of our hybrid method is directly related to the derivability of a set of queries: higher derivabilities indicate better performance.

Looking at the performance with regard to the various cache sizes, we observe that when the cache is small, our hybrid method, being unable to store enough materialized views, underperforms when compared to sequential execution. However, as we cross a cache size and derivability threshold, a critical point (256MB in our case) is reached where sufficient space as well as favourable queries are available, which boosts the performance. When the cache size and the derivability is maximum, our hybrid method executes queryloads twice as fast as sequential execution and SSE.

As mentioned in Section 4.2.2, smaller caches benefit more from sharing sub-expressions, and larger caches from reusing materialized views. However, this is only applicable in the context of our hybrid MQO system, as there is no noticeable improvement in performance when queries are executed using the basic SSE and MVR approaches.

Further analyzing queries on the basis of their operators suggests that filter queries, on average, show a greater variation in performance compared to joins and aggregates. This is likely due to the complexity of the operators. As complex operators take longer to process, sequential execution is at a particular disadvantage because it processes queries in isolation, resulting in repeated processing of same operators. Queryloads containing joins and aggregates show least variation, but also show lower average performance compared to queryloads with only joins or only aggregates. Moreover, our hybrid MQO method performs efficiently for queries that contain complex operators such as joins and aggregates.

As a whole, our hybrid method shows a speed-up of 2x when compared to sequential execution, for larger caches and higher derivabilities. The size of the cache plays an important role but offers diminishing returns after a certain threshold (256MB in our case), which is dependent on the size of the queries.

# 5 Related Work

This section weighs our hybrid Multi-Query Optimization (MQO) method by comparing it with various approaches that have a similar goal as ours: to find effective and efficient methods of optimizing multiple queries in a shared setting.

Michardi et al. [MCM19] propose a batched MQO approach that uses common sub-expressions to improve query execution efficiency in data-intensive workloads, while simultaneously taking into account the memory constraints of the system. The technique that they propose creates covering expressions (CEs) from identified sub-expressions, using which a global shared execution plan is generated. In our approach too we generate covering expressions from multiple sub-expressions, but the two major differences between our approach and the one proposed by [MCM19] are that the latter uses dynamic programming to select from a list of candidate shared plans and that it works only for a batch of queries.

Similarly, Silva et al. [SLZ12] propose a batched MQO technique in the context of cloud query processing systems. Similar to our approach, theirs also produces a shared optimal execution plan that is locally non-optimal. The optimization is done in phases where earlier phases contribute towards greater efficiency, while also being quicker than the later ones. Their implementation, unlike ours, is carried out on SCOPE, which is an optimizer and a scripting language with a query syntax similar to that of SQL.

For realtime database systems, one of the most commonly used technique, as we have seen, is to reuse materialized views to enable optimization. Bachhav et al. [BKS21] implement such a MQO approach, also for cloud query processing systems, that stores and reuses materialized views from earlier queries to process later queries.

Perez and Jermaine [PJ14] design a novel real-time query optimizer, termed Hawc (*History aware cost-based optimizer*), that uses history information and cached materialized views to create an efficient query plan for suitably derivable queries. In Hawc, cached materialized views are ranked based on their utility towards newly submitted queries, thus ensuring that irrelevant views are not indefinitely retained. When the size of the view cache reaches a threshold, irrelevant views are purged to make space

for newer, more fitting views. Being a realtime MQO approach, our hybrid MQO method shares some rough similarity in that regard, but the major difference is in caching. The authors have proposed a caching technique that values materialized views based on its benefit to the historical queryload. Our hybrid method could benefit from this type of selective caching.

A second common approach used in implementing MQO in realtime query processing is caching and reusing Intermediate Results (IR) that are generated as a by-product of query execution. This method suffers from a lack of universal applicability; it can only be implemented in certain query execution models, as outlined in Section 2.1.3. Dursun et al. [DBCK17] propose a new main memory database system, *HashStash*, that reuses intermediate results in both batched and realtime contexts. In their approach, a greater emphasis is placed on studying the role of hash tables in hash-based operator implementations.

Other noteworthy approaches to MQO include Ivanova et al.'s [IKNG10], where MonetDB, which uses the operator at a time execution model, is used as a foundation for building a query optimizer that reuses the generated intermediate results. The performance gained is shown to be substantial even for small sub-queries.

Goldstein and Larsen [GL01] propose and design a fast and scalable approach to determine whether and to what extent the results of a query can be derived from a materialized view. Their approach is also used as the basis for materialized view substitution algorithms in Apache Calcite.

Roy et al. [RSSB00] demonstrate that a heuristic greedy batched MQO approach can offer a good performance, provided some pre-processing is performed. Roy et al. [RRS+00] also propose a realtime MQO technique, termed *Exchequer*, that reuses intermediate results. This method considers only a subset of operators and their properties when optimizing queries, but it exhibits performance gains nonetheless. Zhou et al. [ZLFL07] design a realtime MQO approach that can also efficiently optimize nested queries.

In addition to the above mentioned MQO approaches, some researchers have also studied more specific aspects of MQO. Makreshanski et al. [MGAK18] study the effect of performing joins of hundreds of queries in a distributed fashion across many cores. Jindal et al. [JKRP18] propose an approach, BIGSUBS, that is specifically designed to identify common sub-expressions from a huge volume of queries in an efficient manner. In kind, Ge et al. [GYG+14] propose a Lineage-Signature method to identify common sub-expressions that initially extracts lineage information from queries and then gauges similarity of the expressions. Their findings demonstrate that lineage information allows for quick and efficient selection of common sub-expressions. Jonathan et al. [JCW18] study the effect of executing queries in a shared computational framework over a Wide Area Network (WAN

# 6 Conclusion

In this thesis, we have proposed a hybrid Multi-Query Optimization (MQO) technique to efficiently optimize and share query execution among many queries. We have looked at common MQO techniques, and their advantages and limitations. And we have shown that by composing existing MQO techniques, we can achieve a query processing system capable of halving the time taken to execute suitable queryloads.

Initially, we considered briefly, how traditional query execution is designed. After the discussion about how queries are transformed into query plans through parsing and optimizing, we looked at how query plans can be further optimized using materialized views before examining MQO in a broader sense.

Our hybrid MQO technique combines Shared Sub-Expressions (SSE) and Materialized View Reuse (MVR) to achieve favorable performance. In SSE, sub-expressions that are common among a set of queries are identified, allowing the creation of composite query plans. These composite query plans are used to extract the results of its constituents. In addition to batching queries, we generate materialized views that are cached and can also be used to retrieve relevant query results. The cache observes a Least Recently Used (LRU) policy for ensuring that irrelevant materialized views are pruned. The combination of batching and materialized view substitution, along with a dedicated cache, forms the basis for our hybrid MQO method.

We have evaluated our hybrid MQO method for different queryloads, cache sizes, and compared the results with different execution strategies. The primary constraints that affect the performance of our hybrid MQO technique are the size of the cache and the derivabililty of the queryload. We have seen, from our evaluations, that higher cache sizes and higher derivabilities directly correspond to better performance of the hybrid system. Whereas queryloads with low derivabilities can cause the hybrid system to expend additional resources managing optimization resulting in execution times greater than those for sequential execution, queryloads that are very similar can observe a substantial increase in performance, with execution times being halved in the best case.

Additionally, comparing our hybrid MQO method to the base MVR and SSE approaches shows the conditions under which they are most effective. Reusing materialized views contributes towards greater gains in performance when the cache is larger. For smaller caches, the majority of performance improvements are a result of sharing common sub-expressions.

Further analysis of the effect of different query operators on the performance suggests that queries containing heavier and more complex operators such as joins and aggregates benefit more from a hybrid scheme of processing than queries that contain filters. In summary, a hybrid MQO technique combining SSE and MVR demonstrated clear advantage in execution of up to 2x speed-up over traditional execution.

# 7 Future Work

Extending the work presented in this thesis can be carried out along two broad axes. Firstly, implementation of our hybrid method can be refined to consider additional aspects and take advantage of further characteristics of the provided queries. For instance, as stated in Section 3.2, our hybrid method supports the most commonly used operators during optimization: selections, projections, natural joins, and aggregates. This can be further expanded to cover more operators such as sorting, ordering, and other types of joins.

Moreover, analysis of different cache replacement policies will help establish its importance in the observed overall performance. In a similar vein, machine learning algorithms can be used to more efficiently select materialized views suitable for caching based on their *reusability*. This metric can be determined using additional meta-information about the queries or from the historical queryload.

The second axis along which this work can be extended is the evaluation of our hybrid method. Our hybrid method can be evaluated with other database systems to gauge its performance across different database architectures. Additionally, a more systematic testing with benchmark queries can be performed to directly compare our hybrid method to similar existing techniques.

As we have seen in Section 4.2.2, executing queryloads with lower derivabilities can provide unexpected results when the size of the materialized view and a corresponding derivable query is in stark contrast. If the materialized view is large and the query extracts a negligible amount of data from it, the observed performance is very poor. Additional testing with other database frameworks is necessary to analyze this effect.

Additionally, testing the effect of the size of the query batch on our hybrid method will also provide insight into the influence of Shared Sub-Expressions (SSE), as it benefits from a higher number of queries in a batch.

# Appendix A

# List of Query Templates

In this section, we present the list of 32 template queries that are used to generate queryloads for the evaluation of our hybrid Multi-Query Optimization (MQO) method against sequential, SSE, and Materialized View Reuse (MVR) approaches. The queries are numbered from $Q_1$ to $Q_{32}$.

The queries are submitted for evaluation in the following order. The order of queries is crucial because any given query can only be derivable if suitable previously executed query results have been cached. For instance, we can see that $Q_4$ has a possibility of deriving partial results — for the *supplier* relation — from $Q_2$ depending on the predicate values of both the queries. In the case that $Q_4$ occurs before $Q_2$, this partial derivability is not possible.

## Multiple Column Filter Queries

$Q_1$:
```
SELECT s_name, s_suppkey, s_acctbal
FROM supplier
WHERE (s_suppkey < %d OR s_suppkey > %d)
   OR  s_acctbal < %.2f
```

$Q_2$:
```
SELECT ps_partkey, ps_suppkey, ps_availqty, ps_supplycost
FROM partsupp
WHERE (ps_availqty < %d AND ps_partkey < %d)
   OR (ps_availqty > %d AND ps_partkey > %d)
```

## Single Column Filter Query

$Q_3$:
```
SELECT ps_partkey, ps_suppkey, ps_availqty
FROM partsupp
WHERE ps_availqty > %d AND ps_availqty < %d
```

# Join Queries

$Q_4$:
```
SELECT s_name, s_suppkey, s_acctbal, s_nationkey
FROM supplier
JOIN nation on s_nationkey = n_nationkey
WHERE (s_suppkey < %d OR s_suppkey > %d) AND s_acctbal < %.2f
```

$Q_5$:
```
SELECT ps_partkey, ps_availqty, ps_supplycost
FROM lineitem
JOIN partsupp on l_partkey = ps_partkey
WHERE ps_partkey < %d AND ps_availqty < %d AND ps_supplycost < %.2f
```

$Q_6$:
```
SELECT s_suppkey, s_name, s_acctbal, c_name, c_acctbal, n_name
FROM nation
JOIN supplier on n_nationkey = s_nationkey
JOIN customer on n_nationkey = c_nationkey
WHERE c_acctbal < %.2f AND (s_acctbal < %.2f OR s_acctbal > %.2f)
```

$Q_7$:
```
SELECT o_totalprice, o_orderkey, o_custkey, c_name, c_acctbal
FROM orders
JOIN customer on o_custkey = c_custkey
WHERE (o_totalprice < %.2f)
   OR (c_custkey > %d AND o_totalprice < %.2f)
   OR (o_totalprice < %.2f)
```

$Q_8$:
```
SELECT ps_partkey, ps_supplycost, ps_availqty,
        s_name, s_suppkey, s_acctbal
FROM partsupp
JOIN supplier on ps_suppkey = s_suppkey
WHERE ps_availqty > %d AND s_acctbal < %.2f AND ps_supplycost > %.2f
```

# Multiple Column Filter Queries

$Q_9$:
```
SELECT l_quantity, l_extendedprice, l_shipdate, l_tax, l_discount
FROM lineitem
WHERE l_discount > %.2f AND l_quantity > %d AND l_tax < %.2f
```

$Q_{10}$:
```
SELECT o_orderkey, o_orderdate, o_orderstatus, o_totalprice
FROM orders
WHERE (o_orderkey > %d AND o_orderkey < %d) OR o_custkey < %d
```

# Aggregate Queries

$Q_{11}$:
```
SELECT count(l_quantity), l_quantity
FROM lineitem
WHERE l_discount > %.2f AND l_quantity > %d AND l_tax < %.2f
GROUP BY l_quantity
```

$Q_{12}$:
```
SELECT count(*), l_quantity, l_discount, l_shipdate
FROM lineitem
WHERE l_discount > %.2f AND l_quantity > %d AND l_tax < %.2f
GROUP BY l_quantity, l_discount, l_shipdate
```

$Q_{13}$:
```
SELECT count(o_custkey), o_custkey
FROM orders
WHERE (o_orderkey > %d AND o_orderkey < %d) OR o_custkey < %d
group by o_custkey
```

$Q_{14}$:
```
SELECT s_name, s_acctbal
FROM supplier
WHERE (s_suppkey < %d OR s_suppkey > %d) OR s_acctbal < %.2f
GROUP BY s_name, s_acctbal
```

$Q_{15}$:
```
SELECT s_name, count(s_name)
FROM supplier
WHERE (s_suppkey < %d OR s_suppkey > %d) AND s_acctbal < %.2f
GROUP BY s_name
```

$Q_{16}$:
```
SELECT ps_suppkey, AVG(ps_availqty)
FROM partsupp
WHERE (ps_availqty < %d AND ps_partkey < %d)
   OR (ps_availqty > %d AND ps_partkey > %d)
GROUP BY ps_suppkey
```

$Q_{17}$:
```
SELECT c_name, avg(c_acctbal)
FROM customer
WHERE (c_acctbal < %.2f AND (c_custkey < %d OR c_custkey > %d))
    OR c_acctbal < %.2f
group by c_name
```

# Join-Aggregate Queries

$Q_{18}$:
```
SELECT n_nationkey, avg(s_acctbal)
FROM supplier
JOIN nation on s_nationkey = n_nationkey
WHERE (s_suppkey < %d OR s_suppkey > %d) AND s_acctbal < %.2f
GROUP BY n_nationkey
```

$Q_{19}$:
```
SELECT n_nationkey, avg(s_acctbal)
FROM supplier
JOIN nation on s_nationkey = n_nationkey
WHERE (s_suppkey < %d AND s_suppkey > %d) OR s_acctbal < %.2f
GROUP BY n_nationkey
```

$Q_{20}$:
```
SELECT ps_partkey, avg(ps_availqty), avg(ps_supplycost)
FROM lineitem
JOIN partsupp on l_partkey = ps_partkey
WHERE ps_partkey < %d AND ps_availqty < %d AND ps_supplycost < %.2f
GROUP BY ps_partkey
```

$Q_{21}$:
```
SELECT l_linenumber, avg(ps_supplycost), max(l_discount)
FROM lineitem
JOIN partsupp on l_partkey = ps_partkey
WHERE ps_partkey > %d AND ps_availqty > %d AND ps_supplycost > %.2f
GROUP BY l_linenumber
```

$Q_{22}$:
```
SELECT s_name, c_name, n_nationkey, avg(s_acctbal)
FROM nation
JOIN supplier on n_nationkey = s_nationkey
JOIN customer on n_nationkey = c_nationkey
WHERE c_acctbal < %.2f AND (s_acctbal < %.2f OR s_acctbal > %.2f)
GROUP BY s_name, c_name, n_nationkey
```

$Q_{23}$:
```
SELECT c_name, avg(c_acctbal), avg(o_totalprice)
FROM orders
JOIN customer on o_custkey = c_custkey
WHERE (o_totalprice < %.2f)
   OR (c_custkey > %d AND o_totalprice < %.2f)
   OR (o_totalprice < %.2f)
GROUP BY c_name
```

$Q_{24}$:
```
SELECT s_suppkey, ps_partkey,
        avg(ps_supplycost), avg(ps_availqty), avg(s_acctbal)
FROM partsupp
JOIN supplier on ps_suppkey = s_suppkey
WHERE ps_availqty > %d AND s_acctbal < %.2f AND ps_supplycost > %.2f
GROUP BY s_suppkey, ps_partkey
```

# Single Column Filter Queries

$Q_{25}$:
```
SELECT p_partkey, p_mfgr, p_type, p_size
    FROM part
    WHERE p_size BETWEEN %d and %d
```

$Q_{26}$:
```
SELECT s_name, s_acctbal
    FROM supplier
    WHERE s_acctbal BETWEEN %.2f and %.2f
```

$Q_{27}$:
```
SELECT ps_partkey, ps_suppkey, ps_availqty
    FROM partsupp
    WHERE ps_availqty > %d and ps_availqty < %d
```

$Q_{28}$:
```
SELECT c_custkey, c_name, c_acctbal
    FROM customer
    WHERE c_acctbal BETWEEN %.2f and %.2f
```

$Q_{29}$:
```
SELECT n_name, n_regionkey
    FROM nation
    WHERE n_nationkey < %d
```

$Q_{30}$:
```
SELECT l_tax, l_quantity
    FROM lineitem
    WHERE l_quantity < %d
```

$Q_{31}$:
```
SELECT l_tax, l_quantity
    FROM lineitem
    WHERE l_tax > %.2f
```

$Q_{32}$:
```
SELECT o_orderkey, o_orderdate, o_totalprice
    FROM orders
    WHERE o_totalprice < %.2f
```

# Bibliography


[BKS21] Archana Bachhav, Vilas Kharat, and Madhukar Shelar. An Efficient Query Optimizer with Materialized Intermediate Views in Distributed and Cloud Environment. *In Tehnički glasnik*, 2021. (cited on page 53)

[BRH+18] Edmon Begoli, Jesús Camacho Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. *In Proceedings of ICMD*, 2018. (cited on pages 36 and 37)

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. *In Proceedings of ACM SIGMOD*, 1998. (cited on page 9)

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing Queries With Materialized Views. *In Proceedings of the 11th ICDE*, 1995. (cited on page 37)

[DBCK17] Kayhan Dursun, Carsten Binnig, Ugur Cetintemel, and TIm Kraska. Revisiting Reuse in Main Memory Database Systems. *In Proceedings of ACM SIGMOD*, 2017. (cited on pages 11 and 54)

[DSRS03] Nilesh Dalvi, Sumit Sanghai, Prasan Roy, and Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 2003. (cited on page 13)

[EN16] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Pearson, 7th edition, 2016. (cited on pages 5, 7, and 11)

[GAK12] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: Killing One Thousand Queries With One Stone. *In Proceedings of the VLDB Endowment*, 2012. (cited on pages 1, 12, and 13)

[GL01] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: a practical, scalable solution. *In Proceedings of ACM SIGMOD*, 2001. (cited on pages 37 and 54)

[GMAK14] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *In Proceedings of the VLDB Endowment*, 2014. (cited on page 12)

[GMUW09] Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom. *Database systems: the complete book*. Pearson Prentice Hall, 2nd edition, 2009. (cited on page 5)

[GS17] Anjana Gosain and Kavita Sachdeva. A Systematic Review on Materialized View Selection. *In Proceedings of the 5th ICFIC*, 2017. (cited on page 16)

[GYG+14] Xing Ge, Bin Yao, Minyi Guo, Changliang Xu, Jingyu Zhou, Chentao Wu, and Guangtao Xue. LSShare: an efficient multiple query optimization system in the cloud. *In Distributed and Parallel Databases*, 2014. (cited on page 54)

[IKNG10] Milena Ivanova, Martin Kersten, Niels Nes, and Romulo Gonçalves. An architecture for recycling intermediates in a column-store. *In Proceedings of ACM SIGMOD*, 2010. (cited on page 54)

[JCW18] Albert Jonathan, Abhishek Chandra, and Jon Weissman. Multi-Query Optimization in Wide-Area Streaming Analytics. *In Proceedings of ACM SIGMOD*, 2018. (cited on page 54)

[JKRP18] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting subexpressions to materialize at datacenter scale. *In Proceedings of the VLDB Endowment*, 2018. (cited on page 54)

[KS17] Tarun Kathuria and S. Sudarshan. Efficient and Provable Multi-Query Optimization. *In Proceedings of ACM SIGMOD*, 2017. (cited on page 15)

[MCM19] Pietro Michiardi, Damiano Carra, and Sara Migliorini. In-memory caching for multi-query optimization of data-intensive scalable computing workloads. *In Proceedings of DARLI-AP*, 2019. (cited on pages 12, 13, 14, and 53)

[ME92] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 1992. (cited on page 8)

[MGAK18] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Many-query join: efficient shared execution of relational joins on modern hardware. *In The VLDB Journal*, 2018. (cited on page 54)

[NBV13] Fabian Nagel, Peter Boncz, and Stratis D. Viglas. Recycling in pipelined query evaluation. *In Proceedings of the 29th ICDE*, 2013. (cited on page 1)

[PJ14] Luis Perez and Christopher Jermaine. History-aware query optimization with materialized intermediate views. *In Proceedings of the 29th ICDE*, 2014. (cited on pages 15, 17, and 53)

[RRS+00] Prasan Roy, Krithi Ramamritham, S. Seshadri, Pradeep Shenoy, and S. Sudarshan. Don't Trash your Intermediate Results, Cache 'em. *In CoRR*, 2000. (cited on page 54)

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *In Proceedings of ACM SIGMOD*, 2000. (cited on pages 11, 13, and 54)

[Sel88] Timos Sellis. Multiple-query optimization. *In Proceedings of ACM SIGMOD*, 1988. (cited on pages 2 and 12)

[SLZ12] Yasin Silva, Paul-Ake Larson, and Jingren Zhou. Exploiting Common Subexpressions for Cloud Query Processing. *In Proceedings of the 28th ICDE*, 2012. (cited on pages 12, 13, 14, and 53)

[ZLFL07] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. *In Proceedings of SIGMOD*, 2007. (cited on page 54)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 4 May 2022