What Happens When Two Multi-Query Optimization Paradigms Combine?

A Hybrid Shared Sub-Expression (SSE) and Materialized View Reuse (MVR) Study

Bala Gurumurthy[1](✉), Vasudev Raghavendra Bidarkar[1], David Broneske[2], Thilo Pionteck[1], and Gunter Saake[1]

[1] Otto-Von-Guericke Universität, Magdeburg, Germany
{bala.gurumurthy,vasudevraghavendra.bidarkar,thilo.pionteck, gunter.saake}@ovgu.com

[2] German Centre for Higher Education Research and Science Studies, Hannover, Germany

Abstract. Querying in isolation lacks the potential of reusing results, that ends up wasting computational resources. Multi-Query Optimization (MQO) addresses this challenge by devising a shared execution strategy across queries, with two generally used strategies: *batched* or *cached*. These strategies are shown to improve performance, but hardly any study explores the combination of both. In this work we explore such a hybrid MQO, combining batching (Shared Sub-Expression) and caching (Materialized View Reuse) techniques. Our hybrid-MQO system merges batched query results as well as caches the intermediate results, thereby any new query is given a path within the previous plan as well as reusing the results. To study the influence of batching, we vary the factor - `derivability` - which represents the similarity of the results within a query batch. Similarly, we vary the cache sizes to study the influence of caching. Moreover, we also study the role of different database operators in the performance of our hybrid system. The results suggest that, depending on the individual operators, our hybrid method gains a speed-up between 4× to a slowdown of 2× from using MQO techniques in isolation. Furthermore, our results show that workloads with a generously sized cache that contain similar queries benefit from using our hybrid method, with an observed speed-up of 2× over sequential execution in the best case.

Keywords: Multi-Query Optimization · batched query execution · materialized view reuse

1 Introduction

In many OLAP instances, users submit multiple queries to a DBMS in a short time [GHB+20]. These queries are similar, such that processing them together might save processing time [BKSS18]. To illustrate, Fig. 1 shows the frequency of TPC-H tables fetched in the 22 benchmark queries. As we can see, multiple queries access the same table (e.g., 15 queries access lineitem) that combing these queries can avoid redundant table scans. Coming up with such a common execution plan is the key goal of multi-query optimization (MQO).

MQO processes a query set commonly in two ways: shared sub-expressions (SSE) [SLZ12] and materialized view resolution (MVR) [PJ14]. The former devise a single execution plan for a batch of queries, while the latter optimizes queries individually on the fly. Though these techniques are beneficial, they also have key disadvantages: SSE does not persist results, and every batch is given a new execution plan. On the contrary, MVR persists the query results but lacks a common execution plan for a query batch. However, a hybrid of these techniques can avoid both their disadvantages.

Hybrid MQO Technique: In a gist, our hybrid MQO comes with a common query plan for a query batch and stores its intermediate results. Whenever a new query is introduced, we reuse both the existing execution plan and the stored results. Any new results are persisted replacing existing ones using caching techniques. Thus in this work, we successfully combine common query planning of shared sub-expression with results reuse from materialized views. More details about our hybrid MQO technique are discussed in Sect. 4.

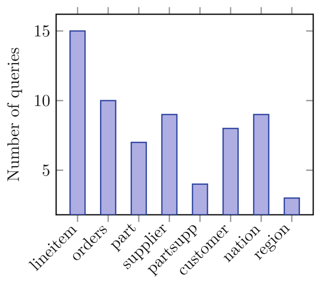


Fig. 1. No. of times TPCH tables accessed in benchmark queries

Due to its flexibility, the hybrid MQO can work in both batched and real-time contexts. Accordingly, we examine the performance of our hybrid MQO w.r.t. sequential, batched, and real-time execution of the queries under different evaluation settings. Our performance evaluation shows a consistent performance improvement which in the best case reaches up to 2× over traditional methods. Specifically, we consider the following aspects when designing and evaluating our hybrid method:

- Workload impact: The effect of various database operators on the hybrid method.
- caching impact: The effect of different cache sizes on the hybrid method.
- Batching impact: The effect of query similarities on the performance of our hybrid method.
- Baseline comparison: The viability of a hybrid method using performance comparison against SSE, MVR, and sequential isolated execution.

Our work is structured as follows: In Sect. 2, we describe related existing MQO techniques. Next, in Sect. 3, we briefly introduce background concepts for MQO. We explain the hybrid multi-query optimizer in Sect. 4 and evaluate it against existing approaches in Sect. 5. Within Sect. 5, we first describe the dataset used, followed by our performance evaluation. Finally, our findings and conclusion are summarized in Sect. 6.

2 Related Work

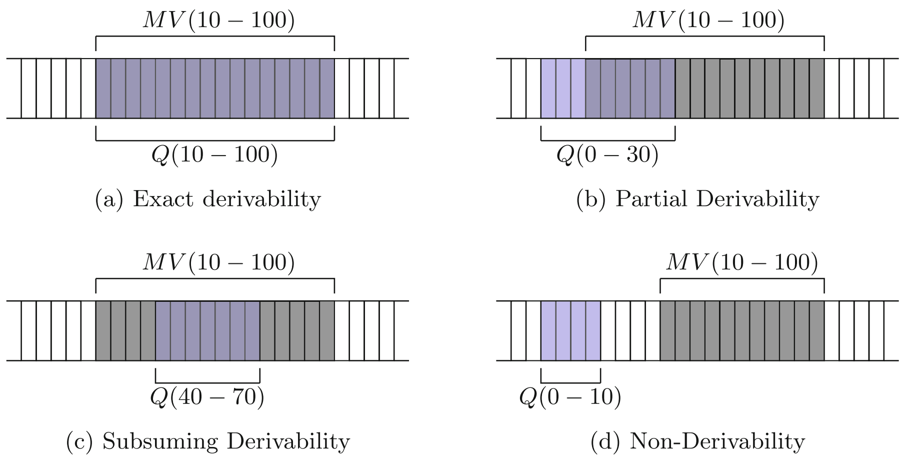
This section weighs the hybrid MQO with various other existing MQO approaches. A closely related work - OLTPshare by Rehrmann et al. [RBB+18] - proposes result sharing within a batch of OLTP queries. However, in our work, we analyze the sharing potential among OLAP workloads. The main goal of SSE is to identify covering expressions, hence many earlier SSE solutions used dynamic programming to identify them [MCM19]. A cloud-based solution is proposed by Silva et al. [SLZ12]. Unlike these, we statically identify SSE from a batch of queries. In the case of MVR, Bachhav et al. [BKS21] proposes a cloud-based solution to process query batches. Perez and Jermaine [PJ14] design Hawc (*History aware cost-based optimizer*), that uses the historical query plans to derive results. However, unlike our approach, these approaches don't batch upfront a set of queries before execution. In addition to the above-mentioned MQO approaches, many works improve the efficiency of these MQO approaches. Makreshanski et al. [MGAK18] study the effect of joins in hundreds of queries. Jindal et al. [JKRP18] propose BIGSUBS - designed to efficiently identify common sub-expressions. Similarly, Ge et al. [GYG+14] propose a lineage-signature method for common sub-expressions. Jonathan et al. [JCW18] study the effect of executing queries in a shared computational framework over a wide area network (WAN). All these approaches complement the hybrid MQO by enhancing its performance.

3 Background

MQO leverages the similarities across queries to avoid re-computation. Specifically, MQO approaches derive the result for a given query from the results of its predecessor. Depending on the similarity across queries, we have varying types of derivability. Likewise, we have varying MQO approaches depending on their characteristics. In this section, we give an overview of derivability as well as the MQO approaches used in our hybrid system.

Derivability: *Derivability* quantifies the amount of results that can be derived from an existing result set [DBCK17,RSSB00]. As illustrated in Fig. 2, there are four general types of derivability: *exact* - where the result is the exact copy (Fig. 2-a), *partial* - only a subset of results is present (Fig. 2-b), *subsuming* - the current result is part of the previously computed ones (Fig. 2-c) and *none* - where no results can be derived (Fig. 2-d). These types are illustrated in Fig. 2.

Please note that although the given example seems to be only applicable to selection clauses, the same applies to more complex queries such as aggregations or joins. In the next section, we will take a broader look at MQO, examining the two important types of techniques that are used in executing queries mutually.



(a) Exact derivability (b) Partial Derivability (c) Subsuming Derivability (d) Non-Derivability

Fig. 2. Types of derivability

MQO Approaches
As we have mentioned, the goal of MQO is to produce an efficient execution plan for multiple queries, irrespective of the performance of individual ones [Sel88]. In In this section, we briefly explain the working of the two MQO approaches considered.

3.1 Batching - Shared Sub-expression

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 optimization more sound [GMAK14, MCM19, SLZ12]. An obvious problem with this approach is that the time taken to create batches is inversely related to the optimization performance. Unfortunately, there is no ubiquitous solution to what the batching time should be, primarily because this is context-dependent. Alternatively, a single query can be split 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 shared sub-expressions (SSE).

In 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. Naturally, as the number of queries in a set increases, the possibility of finding suitable common sub-expressions also increases.

3.2 Real-Time - Materialized View Reuse

In real-time processing, the queries are submitted to the database system as they occur [PJ14]. Here, the optimized query execution plan can be viewed as a growing window. Hence, the advantage of such an approach over batching is its promptness in delivering results. Materialized View Reuse (MVR) does this by caching previous results.

Even though real-time processing solves the problem of delayed responses posed by batched processing, it has a few problems of its own. Firstly, when processing queries in real-time, i.e., separately, we have only fewer avenues for optimization – as complete information about upcoming 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.

So far, we studied the main techniques that encompass MQO. In the next section, we explore a hybrid MQO that places itself in the midst of the two approaches. In this way, we can leverage the benefits of both techniques while minimizing the limitations.

4 SSE-MVR Hybrid Multi-query Optimization

To support a hybrid MQO, certain extensions are needed in a traditional database system. While SSE needs batching as well as generating a shared execution plan, MVR needs access to the stored materialized views while preparing an optimal execution plan. Hence, we form the hybrid of SSE & MVR with: a query batcher, a substituter, and a cache. The overall structure of the hybrid system is shown in Fig. 3.

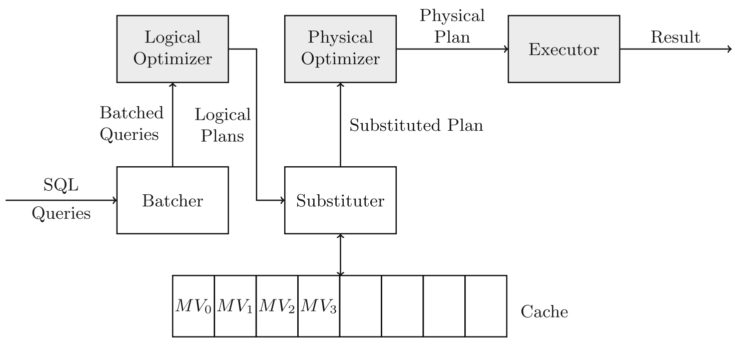


Fig. 3. Components of hybrid MQO system. The blocks colored in gray belong to a conventional database system, whereas the ones in white are exclusive to the hybrid MQO. (Color figure online)

The execution flow in the system is as follows. First, we start batching queries to develop a common query plan. We perform SSE on the given batch of queries

- pruning the redundant operators, identifying the type of derivability possible across queries, etc. Next, such a composite plan is then optimized using an existing traditional optimizer, further simplifying the query plan. The plan is then optimized for MVR based on the cached views from previous runs. If no suitable view is present, we create one from the current run and cache it for future use. In the following sections, we detail the working of the individual components of the hybrid MQO.

4.1 Query Batching

The basic necessity for creating a shared plan across two queries is to have at least a common table scan. For these queries, we then simplify their filter predicates. Similarly, in the next step, we simplify their projection. Let us consider the below queries to be batched. Then, the steps for batching are as follows.

Generating Composite Clauses: Given a batch of queries with common table scans, we start with pruning the ones with common `WHERE` clauses. However, before combining their `WHERE` clauses, we must ensure that the clauses are in their canonical form .i.e in their Conjunctive Normal Form (CNF). Next, we combine these predicates with the `OR` clause. Though such a combined `WHERE` clause might look complex and might even have redundant predicates, the logical optimizer in any traditional DBMS must be able to resolve them easily. Let us consider the queries **Q1 & Q2** below as our running example. We already see that both access lineitem table, therefore the combined predicate simplified with the `OR` clause would look as in Q_{12}. A drawback is that the composite clause may be lengthy because it may contain redundant information. These are simplified using predicate covering.

Q_1: **SELECT** l_quantity, **COUNT**(l_quantity)
FROM lineitem
WHERE l_discount > 0.06
AND l_quantity < 10
AND l_tax > 0.01
GROUP BY l_quantity

Q_2: **SELECT** l_quantity, l_discount, **COUNT**(*)
FROM lineitem
WHERE l_discount > 0.08 **OR**
(l_quantity < 15 **AND** l_tax > 0.03)
GROUP BY l_quantity, l_discount

Q_{12}: **SELECT** ...
FROM lineitem
WHERE (l_discount > 0.06 **OR**
l_discount > 0.08)
AND (l_quantity < 10 **or** l_quantity < 15)
AND (l_tax > 0.01 **or** l_tax > 0.03)
...

Predicate Covering: Even after eliminating irrelevant predicates, we are often left with some predicates that can be simplified according to their access patterns. Our running example after CNF has redundant predicates over the search columns that are to be simplified. For example, `l_quantity < 10 OR l_quantity < 15` can be simplified as `l_quantity < 15` without any change in the output. Similarly, we can also simplify `l_tax < 0.03 OR l_tax > 0.01` fetching all values above 0.01. Once a composite predicate clause is generated, we have to focus on combining operations on projection clauses like aggregations.

Generating Composite Aggregations: Similar to predicates, we can only combine queries that aggregate over a common column. Our example has Q_1 grouping over the *l_quantity* column whereas Q_2 groups on the columns *l_quantity* and *l_discount*. To combine them, we *de-aggregate* the queries, so that the columns are fetched, which are then individually aggregated. Applying this, the de-aggregated query would look like Q_{12D}. Finally, from this de-aggregated query, we can channel the results to compute aggregates based on the individual query.

Q_{12D}: **SELECT** l_quantity, l_discount
FROM lineitem
WHERE l_discount > 0.08
AND l_quantity < 15
AND l_tax > 0.01

Q_1: **SELECT** l_quantity, **COUNT**(l_quantity)
FROM Q_{12D}
WHERE l_discount > 0.06
AND l_quantity < 10
GROUP BY l_quantity

Q_2: **SELECT** l_quantity, l_discount, **count**$(*)$
FROM Q_{12D}
WHERE l_tax > 0.03
GROUP BY l_quantity, l_discount

Though the queries seem to do redundant computations at first, we now can use the composite query to cache the results that can be used in the subsequent queries as well. The steps for MVR with such a composite query are given below.

4.2 Materialized View Reuse

At this optimization stage, the shared query plan from the batched queries is taken as input and compared with existing materialized views for result substitution. We essentially pick from the cached materialized views those that are useful in executing a batched query plan and then substitute the relevant parts of the query plan to utilize the materialized view instead of the database.

Of course, finding out whether a query is derivable requires storing materialized views. We will discuss the specifics of how the cache is designed to this end in Sect. 4.2. The process of reusing materialized views for `JOIN` operator is quite different. For SPSVERBc6s, we need the 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. Finally, since there is only a limited memory available for caching materialized views, we have to regularly evict the materialized views that are not frequently accessed.

Materialized Views Cache: Materialized view cache defines the way to access and maintain the cached views. When the cache reaches a threshold (80% of the total size in our case), a clean routine is called to remove irrelevant views from the cache. Our clean routine follows LRU eviction, such that older queries are evicted whenever a new one has to be inserted. Ultimately, using LRU we indirectly enforce a sliding window in our query batch.

We have so far looked at the individual components that constitute 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 a relevant example.

4.3 System Integration

In this section, we briefly explain the interaction between all these components. Similar to the example above, let us assume the below batch of queries $Q_1 - Q_6$.

Q_1: **SELECT** s_name, s_suppkey, s_acctbal
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, l_partkey
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_discount < 0.05
GROUP BY l_quantity

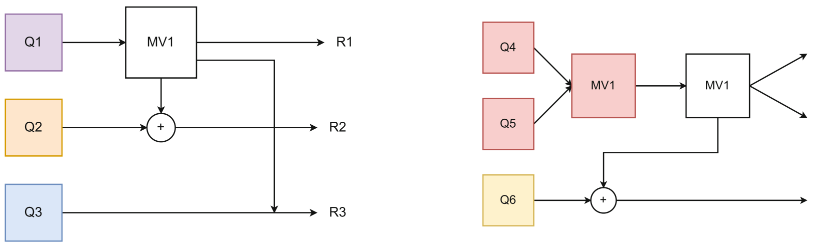


Fig. 4. Functional example of our hybrid MQO

Firstly, the batcher segregates queries that can be batched together. In our case, Q_4 and Q_5 can be batched as they both access `lineitem`. For others, we still have the possibility of achieving performance gains through materialized view substitution.

Next, the optimizer checks for any existing materialized view that can be used to execute Q_1. Since our cache is initially empty, a new view representing Q_1 is generated (MV_1 in Fig. 4). This view can be now reused for Q_2, as `s_suppkey` column is now present in MV_1. A similar case is also applicable for Q_3, Q_4 ,and Q_5.

Finally, to execute Q_6, the optimizer goes through similar steps as for the previous queries. Q_6 is a join of the relations *lineitem* and *partsupp* and the information about *lineitem* can be obtained from the materialized view MV_{4+5}. The tuples for the relation *partsupp* are fetched from the database, and then the join is performed. Now that we have seen the working of our hybrid MQO, in the following section, we evaluate the performance of our hybrid MQO.

5 Evaluation

In this section, we study the performance implications of using a hybrid multi-query optimizer. For our measurements, we extend the existing MVR in Apache calcite (using PostgreSQL plugin) with simple static batching as explained in Sect. 4.1[1]

Evaluation Setup
Due to missing OLAP benchmarks for batching, we use the TPCH dataset with scale factor 5 and custom-built queries. We have 320 such custom queries generated from 32 templates derived from the existing TPCH query set. We generate these queries based on three criteria: query type, result size, and derivability. The detail of the query split-up is shown in Fig. 5.

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

Fig. 5. Composition of query templates

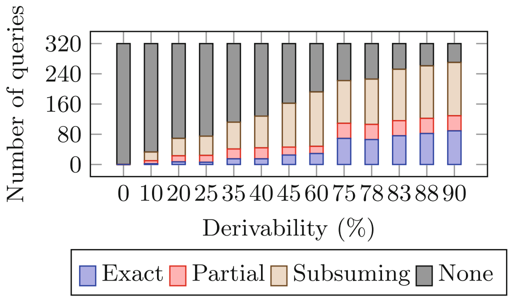


Fig. 6. Proportion of different derivabilities in generated query loads

Impact of Derivability: Furthermore, queries belonging to different derivability types (c.f. Sect. 3) also impact the overall execution. Therefore, it is necessary to ascertain the type of derivability in generated queries. To this end, we plot in Fig. 6 the split-up of different derivability types in a generated query load. As the chart shows, we keep exact derivability at a minimum, with most of the derivability from either partial or subsuming. Now using these queries, in the subsequent sections, we study the performance of hybrid optimization.

Our evaluation uses a Google Cloud - E2-Highmem - instance (with Intel Skylake) with a storage of 100 GB and main memory of 32 GB – all data is stored in main memory.

5.1 Performance Analysis of the Hybrid MQO

In this section, we present our findings in three parts. In Sect. 5.1 we study how different derivabilities and cache sizes influence the execution times of different approaches. Following this, in Sect. 5.1 we compare the performance gains

[1] Code is available here: https://github.com/vasudevrb/mqo.

and losses of different execution strategies over the whole range of tested query loads. Finally, Sect. 5.1 studies the impact of different relational operators on the performance.

The Effect of Derivability and Cache Size: Figure 7 depicts the time to execute different query loads with varying cache sizes. We vary the cache sizes from 4 MB to 2 GB to study its performance impact.

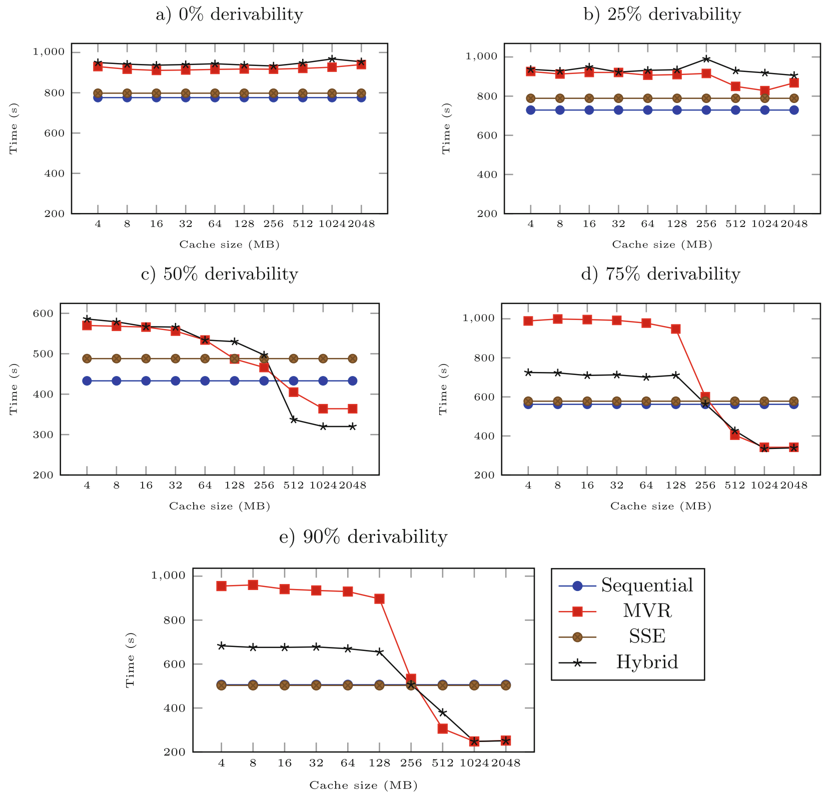


Fig. 7. Execution times of different execution strategies

Foremost, we can see that caching has no effect on sequential and SSE execution as they are independent of cache sizes. For low derivabilities (less than 25%) MVR & hybrid versions perform worse than sequential and SSE. Such poor performance is due to the time spent on creating, caching, and probing materialized views in addition to the normal execution time. However as the derivability increases there is a drastic reduction in the execution times for MVR and hybrid executions. At this point, an adequate amount of views are stored to aid the subsequent queries. This pattern of results holds for all higher derivabilities.

Finally, when derivability increases beyond 50%, we see the performance disparity between MVR and the hybrid method increase – almost a factor of 2.

In this section, we have seen how varying derivabilities and cache sizes influence execution times. Since different query loads contain fundamentally different queries, an absolute comparison of the execution times for different derivabilities is ineffective. So in the next section, we compare the performance of our hybrid system across different derivabilities.

Performance Comparison of Execution Strategies: Figure 8 depicts heatmaps of performance gain/loss of our hybrid MQO method compared to sequential, SSE, and MVR, respectively.

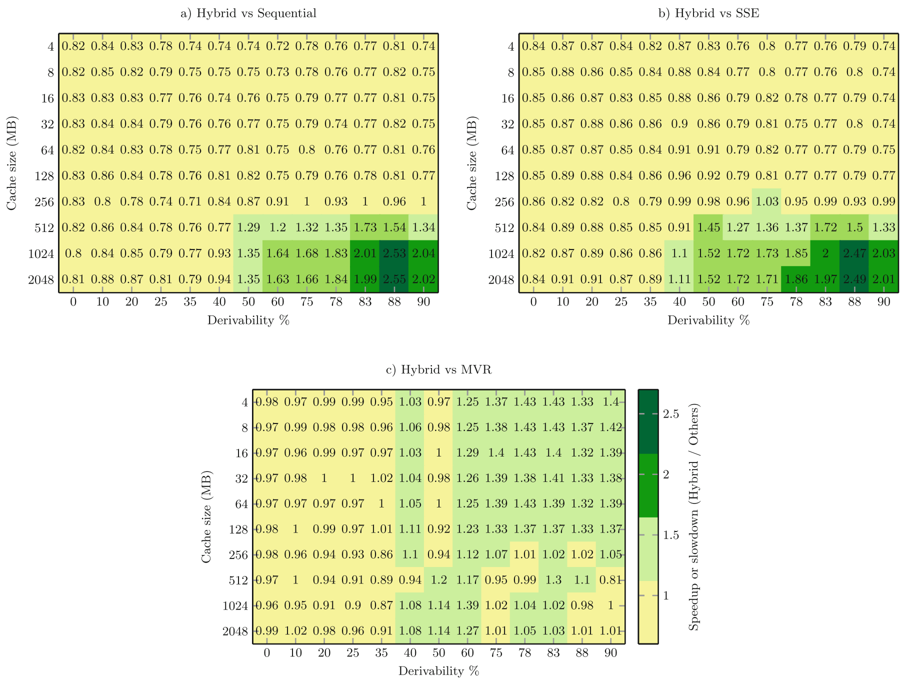


Fig. 8. Speed-up or slow-down from our hybrid mechanism in comparison with baselines

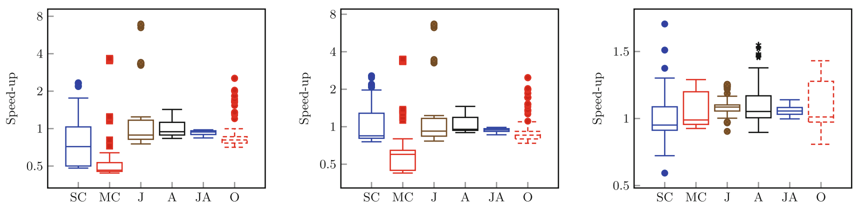
From the heatmaps, our hybrid method is approximately 25% slower for lower derivabilities regardless of cache size. However around 50% derivability, we achieve a 25%–30% gain in performance. This gain keeps increasing, such that with a 90% derivable query load, we are twice as fast as sequential execution.

In some cases, we can observe that having a larger cache harms the performance of our hybrid system. Upon closer inspection, this issue seems unique to Apache Calcite and its handling of materialized view substitutions. Specifically, calcite cycles through the entire materialized view even with 0% derivable queries, degrading performance.

Furthermore, we can observe with high derivability, the performance of our hybrid method exceeds the baseline of the basic MVR approach (Fig. 8-c) when the cache is small. This observation suggests that sharing sub-expressions contributes to this increase in performance. Similarly, for larger caches, the hybrid method is much more efficient when compared to SSE, as shown in Fig. 8-b, implying that the efficiency is a result of materialized view reuse.

Overall, we see a 2x speed-up over sequential and SSE when the queries are derivable, and the cache is large, which shows the full benefit of our approach.

Impact of Query Types: Finally in this section, we study the impact of operators with MQO techniques. The overall speed-up/slow-down is given in Fig. 9. In an overview, we see low derivabilities severely diminish the performance of our hybrid method, whereas high derivabilities enhance performance. A detailed description of the results is given below.



(a) Hybrid vs. sequential execution (b) Hybrid vs. SSE execution (c) Hybrid vs. MVR execution

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

Fig. 9. Relative performances of individual query types

We observe large fluctuations for filter queries, with speed-ups around 0.5 and 2, respectively. Multiple-column filter queries perform poorly than single-column filter queries, with most multi-column filter query loads being slower to execute with our hybrid method than with sequential ones. The performance deterioration is mainly due to the generation of predicate results from an existing materialized view. However, we observe operators such as joins and aggregates demonstrate improved performance than filters. At best, query loads containing exclusively joins show a speed-up of over 6x. As a whole, mixed query loads with filters, joins, and aggregates show insignificant speed-up. As 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. Altogether, under high derivable workloads, our hybrid method performs much better compared to sequential (2×) and SSE (2×) executions than MVR (1.4×). This mirrors our already drawn conclusions.

5.2 Discussion

Since the hybrid system has both batching and MVR, it has good performance with large cache and high derivability - which is refected in our results. As we know from our query load (c.f. Fig. 6) that exactly derivable queries are not 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. Looking at the performance with regard to the various cache sizes, a certain threshold (256 MB in our case) must be crossed to see the benefits of the hybrid execution. When the cache size and the derivability is maximum, our hybrid method executes query loads twice as fast as sequential execution and SSE. However, even with smaller caches we still get a considerable benefit from the SSE part of the system.

Finally, with different database operators, we see that filter operations have a large performance variation than joins and aggregates. Additionally, query loads containing joins and aggregates show the least variation but also show lower average performance compared to query loads with only joins or only aggregates.

As a whole, our hybrid method shows a speed-up of 2× 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 (256 MB in our case), which depends on the query size.

6 Conclusion

In this paper, we have proposed a hybrid MQO technique that merges shared Sub-Expression (SSE) and Materialized View Reuse (MVR). 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 query loads. Our SSE generates a composite query plan whose results are then persisted using MVR, which uses LRU as the cache eviction mechanism. We have evaluated our hybrid MQO method for different query loads, cache sizes, and compared the results with different execution strategies. from our evaluations, we see that high cache-size & derivabilities directly correspond to better performance of the hybrid system. Whereas low derivabilities can cause the hybrid system to expend additional resources managing optimization thereby increasing execution time. Additionally, comparing the hybrid MQO to MVR and SSE approaches shows that it can adapt to the workload. It reuses materialized views when the cache is larger, while with smaller caches, the performance gain is from SSE. Further, analyzing the effect of database operators suggests that 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 demonstrates a clear advantage of up to 2x speed-up over traditional methods.

References

[BKS21] Bachhav, A., Kharat, V., Shelar, M.: An efficient query optimizer with materialized intermediate views in distributed and cloud environment. In: Tehnički glasnik (2021)

[BKSS18] Broneske, D., Köppen, V., Saake, G., Schäler, M.: Efficient evaluation of multi-column selection predicates in main-memory. IEEE Trans. Knowl. Data Eng. **31**(7), 1296–1311 (2018)

[DBCK17] Dursun, K., Binnig, C., Cetintemel, U., Kraska, T.: Revisiting reuse in main memory database systems. In: Proceedings of ACM SIGMOD (2017)

[GHB+20] Gurumurthy, B., Hajjar, I., Broneske, D., Pionteck, T., Saake, G.: When vectorwise meets hyper, pipeline breakers become the moderator. In: ADMS@ VLDB, pp. 1–10 (2020)

[GMAK14] Giannikis, G., Makreshanski, D., Alonso, G., Kossmann, D.: Shared workload optimization. In: Proceedings of the VLDB Endowment (2014)

[GYG+14] Ge, X.: LSShare: an efficient multiple query optimization system in the cloud. In: Distributed and Parallel Databases (2014)

[JCW18] Jonathan, A., Chandra, A., Weissman, J.: Multi-query optimization in wide-area streaming analytics. In: Proceedings of ACM SIGMOD (2018)

[JKRP18] Jindal, A., Karanasos, K., Rao, S., Patel, H.: Selecting subexpressions to materialize at datacenter scale. In: Proceedings of the VLDB Endowment (2018)

[MCM19] Michiardi, P., Carra, D. Migliorini, S.: In-memory caching for multi-query optimization of data-intensive scalable computing workloads. In: Proceedings of DARLI-AP (2019)

[MGAK18] Makreshanski, D., Giannikis, G., Alonso, G., Kossmann, D.: Many-query join: efficient shared execution of relational joins on modern hardware. In: Proceedings of the VLDB Endowment (2018)

[PJ14] Perez, L., Jermaine, C.: History-aware query optimization with materialized intermediate views. In: Proceedings of the ICDE (2014)

[RBB+18] Rehrmann, R., Binnig, C., Böhm, A., Kim, K., Lehner, W., Rizk, A.: Oltpshare: the case for sharing in oltp workloads. In: Proceedings of the VLDB Endowment (2018)

[RSSB00] Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. In: Proceedings of ACM SIGMOD (2000)

[Sel88] Sellis, T.: Multiple-query optimization. In: Proceedings of ACM SIGMOD (1988)

[SLZ12] Silva, Y., Larson, P.A., Zhou, J.: Exploiting common subexpressions for cloud query processing. In: Proceedings of the ICDE (2012)