



Optimizing correlated aggregate subqueries in Firebolt

Andrés Senac¹, Alexander Alexandrov ¹, Immanuel Haffner ¹, Zhen Li¹, and Benjamin Wagner¹

Abstract: Queries generated by modern data applications and BI tools often contain repetitive yet similar, possibly correlated subqueries. We present how the Firebolt rule-based optimizer handles a specific class of such queries that contain correlated scalar aggregate subqueries over similar inputs. We do this by crafting a set of rules that work in tandem to achieve the desired end result. We discuss each of these rules in detail and explain why our technique goes beyond the current state of the art.



Keywords: query, optimization, query processing, data warehousing

1 Introduction

Businesses are collecting ever larger and more complex data sets. This has given rise to a wide range of data-intensive applications that allow interacting with data. Internally, these applications are often powered by BI tools such as Looker or Tableau. Customer-facing data applications frequently use custom query generators and visualization layers.

To understand the effects of data applications on the underlying DBMS, let us briefly consider an application that visualizes summaries over large data sets in near real time. Such applications are often interactive – for example, visualizing historical data might allow for changing the presented time range. Additionally, the application can have widgets that impact which data is to be considered. Typically, the application does this by translating each widget into an SQL-level parameter or fragment, incorporating all fragments into a set of queries, submitting the queries to a DBMS, and finally rendering the query results. This means that data applications often act like SQL query generators. To provide an interactive experience, the DBMS has to answer such machine-generated queries in near real-time.

In this work, we present such a machine-generated query, that – to the best of our knowledge – is not optimized well by any major commercial or research DBMS other than Firebolt². The lack of sufficient optimization hinders processing the query in near real-time and renders the data app impractical to work with. This poses a risk for our customers. The goal of this work is to explain how the Firebolt query optimizer optimizes this query so that it can be processed efficiently, and highlight why current state-of-the-art systems fail to do so.

¹ Firebolt Analytics Inc., andres@firebolt.io;
alexander.alexandrov@firebolt.io,  <https://orcid.org/0009-0009-3705-5965>;
immanuel.haffner@firebolt.io,  <https://orcid.org/0009-0003-8796-1129>; zhen@firebolt.io;
benjamin.wagner@firebolt.io

² <https://firebolt.io>

Listing 1 Example query Q computing for each customer the minimum and maximum order price over all orders that are still open.

```

1 select c.*,
2     (select min(price) from orders where status='0' and c.key=ckey) "min",
3     (select max(price) from orders where status='0' and c.key=ckey) "max"
4 from customers c;

```

To that end, our work makes the following contributions. In Sect. 1 we provide an example query that highlights the limitations of current systems. As we cannot disclose customer data, we present an anonymized customer query to drive the discussion. We analyze the query and describe its optimization potential, identifying subquery decorrelation, common aggregates discovery, and redundant join removal as necessary rules to optimize this query. In Sect. 2 we elaborate decorrelation and common aggregate discovery. In Sect. 3 we present redundant join removal. We discuss related work in Sect. 4 and conclude in Sect. 5.

1.1 A Running Example

Listing 1 shows an example query Q that is not optimized well by any major commercial or research DBMS other than Firebolt. The query computes for every customer the minimum and maximum price over all orders that are still open. The minimum and maximum values are computed by two subqueries. Each subquery is correlated through the predicate $c.key=ckey$, where $c.key$ is the correlated, free variable. Each subquery applies the same filter $status='0'$, and each subquery aggregates without a grouping expression.

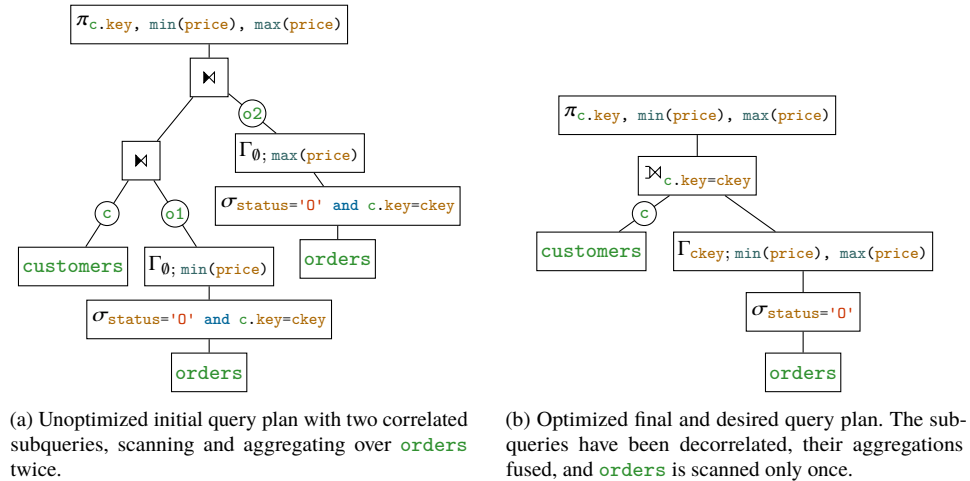


Fig. 1: Two equivalent query plans for example query Q , one before and one after optimization.

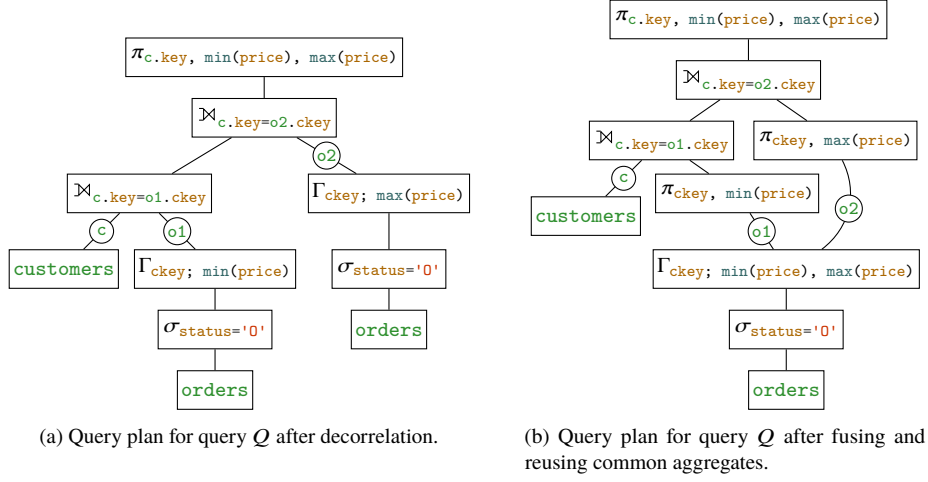


Fig. 2: Incremental optimization of the query plan for query Q .

Translating the SQL statement to an initial query plan can be done easily by following the syntactic structure of the query. The correlated subqueries can be expressed with the help of the *dependent join* operator \bowtie [NK15]. Fig. 1a shows the initial query plan built by the Firebolt optimizer³. Note that the query plan scans the `orders` relation twice, performs the filter and aggregation twice, and then performs two dependent joins. The high degree of redundancy in this plan tremendously impairs performance.

We would like to come up with an equivalent and more efficient plan for query Q . In particular, we shall decorrelate the dependent joins and achieve scanning of and aggregating over `orders` only once. The desired query plan is shown in Fig. 1b. In the next sections, we describe the rules implemented in the Firebolt query optimizer that allow us to transform the initial plan from Fig. 1a into this desired plan.

2 Decorrelation and Common Aggregate Discovery

Fig. 2a shows the query plan after full decorrelation, implemented as described in [GJ01] and [NK15]. Observe that the dependent joins were converted to left outer joins, and the aggregations contain the formerly correlated expression as a grouping key.

Evaluating an aggregate, global or with a `GROUP BY` key, is a costly operation that generally requires a scan over the entire input relation. Conveniently, many of SQL's aggregate functions fall into the class of commutative-associative aggregates. Operators computing such aggregates independently can be fused together into a single operator that computes all

³ Throughout the paper we use the relational algebra operators and notation by Neumann; Kemper [NK15].

aggregates simultaneously in a single pass [AKM19, Sect. 8.1.2]. In our running example, both `min` and `max` are commutative-associative, so we can fuse the two $\Gamma_{ckey; \dots}$ operators from Fig. 2a. The result is the DAG-shaped plan in Fig. 2b.

3 Redundant Join Removal

Finally, to reach our desired final plan in Fig. 1b, we need to remove the redundant join in Fig. 2b. In that plan, one of the two joins is redundant because:

- (i) Both joins are performed on the same grouping key `orders.ckey` that comes from the same subplan shared by `o1` and `o2`. Once we have performed one of the joins, the other join will not duplicate or remove any tuples from the result set.
- (ii) The two subqueries `o1` and `o2` project different sets of columns from a shared subplan. With a minor rewrite, one subquery could provide all columns needed by the operators above. This means that joining the other subquery will not be necessary in terms of columns needed.

If a join does not duplicate or remove tuples and does not provide any new columns, it is redundant. This allows us to remove one of the two joins in Fig. 2b.

Our join removal technique searches for redundant joins by analyzing the join graph. This analysis is performed even in the absence of primary-key or foreign-key constraints. Within a join graph, the process of removing a redundant join corresponds to removing a vertex (which represents a subplan) and an associated edge (which represents the join) from the graph. Upstream expressions that depend on the removed vertex are then rewritten in terms of one of the remaining vertices which acts as a replacing vertex. To account for possible differences between the removed and the replacing vertex, the rewritten expressions need to insert extra filter or projection operators depending on the join type.

We use a more general example to illustrate the redundant join removal process. For simplicity, the discussion here is limited to simple graphs, although the conditions can be extended to work with hypergraphs [MN08]. Consider a similar query on relations `customers(key, name)`, and `maxord(ckey, price)` which stores the largest order per customer.

```
select * from customers c
left join(select * from maxord where price<10) o1 on c.key=o1.ckey
left join(select * from maxord where 5<price<10) o2 on c.key=o2.ckey
```

The logical plan and the join graph are shown in Fig. 3a and Fig. 3b (top), respectively. Searching for the two conditions of redundant joins in Fig. 3a, we see that

- (i) Joining $o2$ does not duplicate or remove any tuples from the $c \text{ left join } o1$ result set because for any value of $c.key$ in the result set, there is at most one value in $o2.ckey$ to join (but customers that have not yet made an order might not be present).
- (ii) In this case, the join graph vertex representing subplan $o1$ already provides all columns needed by the operator above. We need a minor rewrite preserving the outer join semantics, though.

As a result, the join with $o2$ is identified as redundant. The simplified graph is shown at the bottom of Fig. 3b.

Classic approaches detect condition (i) by checking primary-key and foreign-key constraints. Note that we did not assume such constraints to exist when performing redundant join removal. The analysis is done purely based on the semantics of relational operators. Our algorithm is built on top of column provenance analysis. In addition, predicate range analysis is integrated to include the cases in the example from this section. Of course, if primary-key and foreign-key constraints are present, they can be used to derive the first condition, too.

To make sure the simplified query produces a correct result following the outer join semantics, columns from $o2$ that are rewritten in terms of $o1$ have to be guarded by the extra predicates that appear in local filters or in the join condition of the removed outer join $o2$. Generally, the guard expression for a conjunction of extra predicates p and a column x is defined as

$$g(p, x) = \text{case when } p \text{ then } x \text{ else null end}$$

It ensures that columns from the redundant join vertex are correctly set to **null** in order to adhere to the semantics of the removed outer join. In Fig. 3c, p is $5 < o1.price$.

Applying this join removal technique on Fig. 2b with further algebraic simplification, we obtain the final in plan Fig. 1b. In our running example, there are no extra predicates for the removed join, so the guard predicate p to handle outer joins is **true**. The plan therefore is further reduced by removing the now obsolete **case** expressions and χ operator altogether.

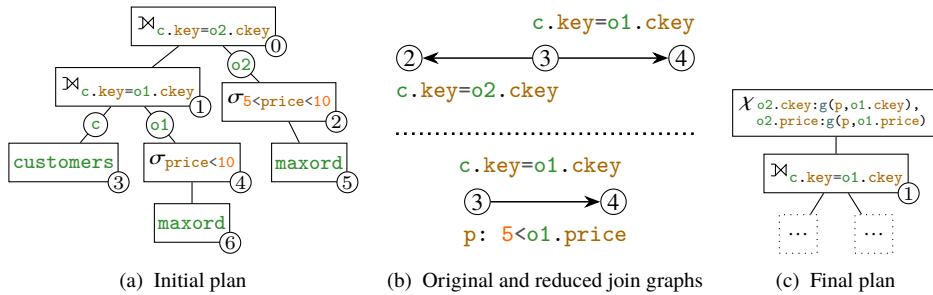


Fig. 3: Redundant Join Removal example.

4 Related Work

Redundant Join Removal was first discussed in the 1980s as one of the heuristics of *semantic query optimization* (SQO) [Ki81; SO89]. In this context, semantic knowledge is usually represented as function-free clauses in predicate logic. Some of the clauses exist in relational databases as constraints, sometimes referred to as functional dependencies.

To the best of our knowledge, the first commercial database that includes a join elimination implementation is IBM DB2 Universal Database [Ch99]. It uses *referential integrity* (RI) constraints to identify redundant joins. Today, the same approach is still being used in modern data warehouses and databases.

However, the sheer amount of data needed to be processed in these systems today can easily cause unbearable overhead in enforcing RI constraints. As a result, many systems do not guarantee RI constraints, even allowing them to be defined. This means the chances of classic join elimination kicking in stay low. From the systems that we surveyed, Materialize⁴ is the only one that attempts to simplify joins based on join graphs and column provenance analysis. However, the redundant outer join case presented here is not currently detected.

5 Conclusion

Modern data applications provide interactive experiences that require the underlying DBMS to serve query results in near-real time. Further, machine-generated SQL queries with similar, possibly correlated subqueries are a natural consequence of how such applications are currently programmed. To meet the demands of their customers, DBMS vendors such as Firebolt therefore have to offer industry-grade query optimizers that can detect and remove naturally occurring redundancies in such machine-generated queries before they are executed.

In this paper, we focused on one such query pattern and demonstrated how the state-of-the-art query unnesting strategy proposed by Neumann; Kemper. “Unnesting Arbitrary Queries” [NK15] at this venue ten years ago leads to a query plan that has obvious redundancies. Even after applying well-known common subplan discovery techniques, none of the commercial or research systems that we surveyed was able to completely eliminate the replicated query fragment. To solve this problem, we proposed a novel technique for detecting and removing redundant joins based on join graph analysis.

The exposition in this paper was limited to a specific class of correlated aggregate subqueries. However, we believe that by augmenting the redundant join removal technique from Sect. 3 with more cases and combining them with other well-known logical simplification rules, a rule-based optimizer can be taught to detect and eliminate redundancies across an even larger, syntactically more varied class of subqueries.

⁴ <https://materialize.com>

References

- [AKM19] Alexandrov, A.; Krastev, G.; Markl, V.: Representations and Optimizations for Embedded Parallel Dataflow Languages. *ACM Trans. Database Syst.* 44 (1), 4:1–4:44, 2019, doi: 10.1145/3281629, URL: <https://doi.org/10.1145/3281629>.
- [Ch99] Cheng, Q.; Gryz, J.; Koo, F.; Leung, T. Y. C.; Liu, L.; Qian, X.; Schiefer, K. B.: Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In: *Proceedings of the 25th International Conference on Very Large Data Bases. VLDB '99*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 687–698, 1999, ISBN: 1558606157.
- [GJ01] Galindo-Legaria, C. A.; Joshi, M.: Orthogonal Optimization of Subqueries and Aggregation. In (Mehrotra, S.; Sellis, T. K., eds.): *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, Santa Barbara, CA, USA, May 21-24, 2001. ACM, pp. 571–581, 2001, doi: 10.1145/375663.375748, URL: <https://doi.org/10.1145/375663.375748>.
- [Ki81] King, J. J.: QUIST: a system for semantic query optimization in relational databases. In: *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7. VLDB '81*, VLDB Endowment, Cannes, France, pp. 510–517, 1981.
- [MN08] Moerkotte, G.; Neumann, T.: Dynamic programming strikes back. In (Wang, J. T., ed.): *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*, Vancouver, BC, Canada, June 10-12, 2008. ACM, pp. 539–552, 2008, doi: 10.1145/1376616.1376672, URL: <https://doi.org/10.1145/1376616.1376672>.
- [NK15] Neumann, T.; Kemper, A.: Unnesting Arbitrary Queries. In (Seidl, T.; Ritter, N.; Schöning, H.; Sattler, K.; Härder, T.; Friedrich, S.; Wingerath, W., eds.): *Datenbanksysteme für Business, Technologie und Web (BTW)*, 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. *Proceedings. Vol. P-241. LNI, GI*, pp. 383–402, 2015, URL: <https://dl.gi.de/handle/20.500.12116/2418>.
- [SO89] Shenoy, S.; Ozsoyoglu, Z.: Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering* 1 (3), pp. 344–361, 1989, doi: 10.1109/69.87980.