Computation Reuse via Fusion in Amazon Athena

Nicolas Bruno
AWS Athena
Amazon
Redmond, US
nicbrun@amazon.com

Johnny Debrodt
AWS Athena
Amazon
New York, US
jdebrodt@amazon.com

Chujun Song
Computer Science Dept.
University of Maryland
College Park, US
cjsong@cs.umd.edu

Wei Zheng
AWS Athena
Amazon
Palo Alto, US
wzheamzn@amazon.com

***Abstract*— Amazon Athena is a serverless, interactive query service that allows efficiently analyzing large volumes of data stored in Amazon S3 using ANSI SQL. Some design choices in the engine, especially those concerning streaming of intermediate results, can result in suboptimal executions for query patterns that have common expressions. In this paper we build upon recent work and introduce new optimizations in Athena that handle some common expression scenarios without materializing intermediate results or duplicating work. We describe commonalities and differences with previous work, and provide experimental results that validate our approach on TPC-DS data.**

Keywords—query optimization, query fusion, Amazon Athena

I. Introduction

Amazon Athena [4] is an interactive query service that makes it easy to efficiently analyze, using ANSI SQL, large volumes of data stored using open formats. Customers do not need to perform Extract-Transform-Load (ETL) operations to ingest data in Athena. Instead, they define schemas pointing to data residing in Amazon S3 [5] and immediately start issuing queries and getting insights[1]. To that end, Athena natively supports a variety of data formats, including CSV (comma-separated values), JSON [6], ORC [7], Avro [8], and Parquet [9], and several compression alternatives, such as Snappy [10], Zlib [11] and Gzip [12].

Athena is serverless, so there is no infrastructure to setup, provision or manage. The control plane allocates compute resources on-demand across multiple availability zones and transparently scales up without the need of complex tuning knobs. Billing follows a pay-as-you-go model, in which customers are charged a fixed amount per TB scanned, without additional storage charges beyond those incurred by S3 and dependent services like AWS Glue [14] and AWS Lambda [15].

The query engine in Athena has its roots in the Presto system [13]. Although the engine diverged over time from the initial Presto fork, it shares various design philosophies and implementation choices that result in a flexible and performant solution. In particular, the query engine is architected to leverage streaming during compilation, scheduling, and execution. For instance, split enumeration (i.e., the task of determining which files to retrieve to evaluate queries) is not synchronous, and the engine starts evaluating portions of the query before enumeration is finished. Also, intermediate shuffle operations, which are required for correctness in a distributed setting, are not materialized to stable storage. Instead, intermediate results are streamed from producer to consumer tasks.

These architectural decisions result in an engine that typically exhibits interactive query performance even for complex queries and large amounts of input data. At the same time, for certain query patterns, they can cause suboptimal executions. A specific scenario, which we focus on in this paper, is that of common subexpressions. Consider, for instance, a variant of query 65 in the TPC-DS benchmark [16]:

```
SELECT s_store_name, i_item_desc, revenue
FROM store, item,
    (SELECT ss_store_sk, AVG(revenue) AS ave
     FROM (SELECT ss_store_sk, ss_item_sk,
                  sum(ss_sales_price) AS revenue
           FROM store_sales, date_dim
           WHERE ss_sold_date_sk = d_date_sk
            AND d_month_seq BETWEEN 1212 AND 1247
           GROUP BY ss_store_sk, ss_item_sk) sa
     GROUP BY ss_store_sk) sb,
    (SELECT ss_store_sk, ss_item_sk,
              sum(ss_sales_price) AS revenue
     FROM store_sales, date_dim
     WHERE ss_sold_date_sk = d_date_sk
        AND d_month_seq BETWEEN 1212 AND 1247
     GROUP BY ss_store_sk, ss_item_sk) sc
WHERE sb.ss_store_sk = sc.ss_store_sk
  AND sc.revenue <= 0.1 * sb.ave
  AND s_store_sk = sc.ss_store_sk
  AND i_item_sk = sc.ss_item_sk
ORDER BY s_store_name, i_item_desc LIMIT 100
```

This query uses the same common block twice in the `FROM` clause (shown in bold above). Due to the streaming nature of the query engine, plans consist of trees of operators without materialization points. Athena thus processes the query by evaluating the common portion twice. This is suboptimal, especially if the duplicated computation is expensive. Due to the pay-as-you-go billing model, this approach also increases the cost to customers when executing these types of queries.

[1] Note that while S3 is the most common data source in Athena, the system supports several other sources via federated connectors.

A common approach to deal with common subexpressions is via spooling [21]. The idea is to evaluate a common computation once and reuse the intermediate results by multiple consumers (inducing DAG-like execution plans). Spooling is a general approach to deal with common subexpressions, and this solution is part of Athena's future roadmap. However, in certain scenarios we can do better than spooling. Some query shapes can be handled in a better way by completely removing multiple instances of the common subquery without the need to store intermediate results. In this way we preserve the streaming characteristics of the engine and return results efficiently while avoiding duplicate computations. As an example, the query above can be rewritten as follows:

```
SELECT s_store_name, i_item_desc, revenue
FROM store, item,
    (SELECT ss_store_sk, ss_item_sk, revenue,
            AVG(revenue) OVER (PARTITION BY
                               ss_store_sk) avgR
     FROM (SELECT ss_store_sk, ss_item_sk,
                  sum(ss_sales_price) AS revenue
           FROM store_sales, date_dim
           WHERE ss_sold_date_sk = d_date_sk
            AND d_month_seq BETWEEN 1212 AND 1247
           GROUP BY ss_store_sk, ss_item_sk) X) Y
WHERE revenue <= 0.1 * avgR
   AND ss_store_sk = s_store_sk
   AND ss_item_sk = i_item_sk
ORDER BY s_store_name, i_item_desc LIMIT 100
```

For this rewrite, we leverage the specific pattern of the query, which aggregates the common expression on `ss_store_sk` and then joins the aggregated result back with the common expression on column `ss_store_sk`. We can achieve the same result by using a windowed aggregation partitioned on `ss_store_sk`, and keeping rows for which `ss_store_sk` is not `NULL` (implicitly done by the subsequent join with table `store`). In this way, we combine a common Join/Aggregation pattern over a common expression into an alternative formulation that evaluates and reads the input only once. This rewrite reduces the query latency by 48% and the amount of data scanned by almost 50%, which translates in lower bills to customers.

The techniques described in this paper can also handle scenarios in which the common expressions are not exactly the same, but can still be rewritten into a fragment that generates a superset of the required rows and columns, and handle the differences via compensating actions. As a simple example:

```
WITH cte as (...complex_subquery...)
SELECT customer_id FROM cte WHERE fname = 'John'
UNION ALL
SELECT customer_id FROM cte WHERE lname = 'Smith'
```

can be rewritten into an equivalent query as:

```
WITH cte as (...complex_subquery...)
SELECT customer_id
FROM cte, (VALUES (1), (2)) T(tag)
WHERE (fname = 'John' AND tag=1)
   OR (lname = 'Smith' AND tag=2)
```

Note that, in general, not all queries with common expressions can be rewritten by eliminating both duplication of work and materialization (the general case should be handled by spooling intermediate results). However, there are several scenarios for which the rewrites shown in this work are applicable. In those cases, the resulting rewrites are more efficient than alternatives that materialize intermediate results, which not only write those intermediates, but need to read them multiple times. For instance, the TPC-DS benchmark improves over 14% when our techniques are enabled, and specifically improves almost 60% for the subset of applicable queries.

It is also worth noting that the same techniques are completely applicable to any database system based on query rewriting, since they do not require new operators or execution models. In fact, other analytic engines within AWS, like Amazon Redshift [22] and Amazon EMR [23], are already incorporating these improvements.

The rest of the paper is structured as follows. In Section II we review previous work on query fusion and adapt it to power our optimizations. In Section III we describe how we implement query fusion in Athena, and in Section IV we introduce optimization rules that rely on query fusion. In Section V we present an experimental evaluation of our techniques on the TPC-DS benchmark, and we conclude in Section VI.

II. Related Work

Big-data query optimizers [13, 17-19] typically borrow and build upon rewrite rules from the database literature. Our approach follows this pattern by leveraging and extending recent work in the literature, and specifically the query fusion work of Blitz [1] and Resin [2].

The Blitz system extends a query optimizer with specific rules that find and substitute subquery patterns with streaming super-operators. Two of these rules are self-joins and self-unions that follow a `GroupBy` on the same input table, and the third pattern is a specialized implementation of a `min` aggregation followed by a join. Blitz transformations are useful whenever applicable, but have some drawbacks. The patterns cover a rather small fraction of queries, and the resulting super-operators do not compose well with each other and therefore cannot be chained together. Our approach is able to cover the optimizations in Blitz without the need of super-operators, as discussed in Section III.

Resin extends the work in Blitz by formalizing the concept of query fusion, which identifies sub-queries computing on overlapping data, and fuses them into a single computation with compensating actions to retrieve the original results. This analysis does not require the subqueries to be syntactically the same nor produce the same output. The fused query can be used once in general via spooling, and in some cases, it can eliminate spooling altogether. For that purpose, Resin introduces new operators (e.g., ResinMap, ResinReduce) to implement query fusion. These operators are simpler to manipulate than the Blitz super-operators, and therefore can be applied to broader scenarios. Our work is based on the Resin work, with the following differences and extensions:

- Our approach handles scenarios that can fully eliminate common subqueries without materializing intermediate results. For that reason, we streamline the discovery of

matches by embedding the process in optimization rules, without using signatures to identify 'far-away' matches that could not be leveraged.
- We introduce optimization rules that leverage query fusion to novel query patterns (e.g., fusing subqueries that differ in a source table, or better handling of *diamond-like* shapes on n-ary operators).
- We implement query fusion without introducing new operators like ResinMap/ResinReduce. Instead, we express the resulting computation using standard relational operators. As a consequence, our new rules can be composed in a better way with existing ones, since there are no new operators to handle. Additionally, it avoids the requirement of additional code generation for new operators.
- We extend query fusion to cover operators that are unique to Athena (e.g., `MarkDistinct`) and in that way, better handle constructs like distinct aggregates.

III. Query Fusion Primitives

We next describe our implementation of query fusion, which serves as building block in the optimization rules we describe in Section IV. Query fusion is a recursive procedure that operates over logical algebraic trees. Specifically, we define a function `Fuse` that takes two input plans and returns either $\perp$ (when fusion is not possible), or a 4-tuple fused result otherwise. If `Fuse(P1, P2) = (P, M, L, R)`, then:

- `P` is the fused resulting plan. The schema of `P` includes all output columns in P1 and, optionally, additional output columns from `P2`.
- `M` is a mapping from the output columns of `P2` to output columns of `P`.
- `L` and `R` are two filter conditions defined over the output columns of `P` to restore `P1` and `P2`, respectively.

Semantically, we can reconstruct `P1` and `P2` as follows:

$P1 = Project_{outCols(P1)}(Filter_{L}(P))$
$P2 = Project_{M(outCols(P2))}(Filter_{R}(P))$
where `outCols(P)` denotes the output columns of plan `P`.

In the rest of this section, we define `Fuse(P1,P2)` based on the shape of the input plans `P1` and `P2`. For now, we require that both `P1` and `P2` have the same root operator, and explain how we relax this assumption in Section III.G. To illustrate each definition, we give samples based on the TPC-DS schema.

A. Table Scans

Fusing two table scans, `Fuse(Scan(T1), Scan(T2))`, succeeds when `T1` and `T2` denote the same table (otherwise, the operation returns $\perp$). The fused result is defined as:

`(Scan(T1), columnMap(T2, T1), TRUE, TRUE)`

where `columnMap(T2, T1)` maps positionally the output columns of `T2` to those of `T1` (note that the engine follows the common practice of assigning new column identities to each instance of the same table). As an example, fusing two query fragments that scan the same table:

```
SELECT i_item_sk AS sk, i_brand AS brand
FROM item

SELECT i_brand AS brand2, i_size AS size
FROM item
```

results in a fused result `(P, M, TRUE, TRUE)` where:

```
P: SELECT i_item_sk AS sk,
          i_brand AS brand,
          i_size AS size
   FROM item
M: brand2 → brand
```

B. Filters

Consider $F1=Filter_{C1}(P1)$ and $F2=Filter_{C2}(P2)$. We first recursively fuse the subplans. If this is not successful, we propagate $\perp$. Else, if `Fuse(P1, P2)=(P, M, L, R)` we define `Fuse(F1, F2)` as[2]:

$(Filter_{C1\ OR\ M(C2)}(P), M, L\ AND\ C1, R\ AND\ M(C2))$

Note that if `C1` is equivalent to `M(C2)`, we simplify the fused result as $(Filter_{C1}(P), M, L, R)$.

As an example, fusing two query fragments that scan the same table with different filters:

```
SELECT i_item_desc
FROM item
WHERE i_category = 'Music' AND i_brand_id > 1000

SELECT i_item_desc
FROM item
WHERE i_category = 'Music' AND i_brand_id < 50
```

results in `(P, ∅, L, R)`, where:

```
P: SELECT i_item_desc FROM item
   WHERE i_category = 'Music' AND
         (i_brand_id < 50 OR i_brand_id > 1000)
L: i_brand_id > 1000 AND i_category = 'Music'
R: i_brand_id < 50 AND i_category = 'Music'
```

C. Projections

Consider $R1=Project_{A1}(P1)$ and $R2=Project_{A2}(P2)$, where `A1` and `A2` are sequences of assignments of expressions to new variables (e.g., `[x:=a+1, y:=b]`). We first recursively compute `Fuse(P1, P2)` and propagate any $\perp$ result.

[2] Strictly speaking, `M` is a map from columns to columns. We abuse the notation for simplicity and reuse `M` to map expressions in the natural way.

Otherwise, if fusing P1 and P2 returns (P, M, L, R), we define the result of Fuse(R1, R2) as:

$$(\text{Project}_{A}(P), M', L, R)$$

where A contains all assignments in A1 and M′ contains all mappings in M. Additionally, for each assignment a:=expr in A2, we check whether A1 contains an assignment of the form b:=M(expr). If that is the case, we add a→b to the resulting mapping M′. Otherwise, we add b:=M(expr) to the resulting assignments A. For instance, if T(a) is a table with a single column a, and T′(a′) denotes another instance of T with a new column a′, the result of fusing $\text{Project}_{x:=a+1}(\text{Scan}(T))$ and $\text{Project}_{y:=a'+1,\ z:=3}(\text{Scan}(T'))$ is:

$$(\text{Project}_{x:=a+1,\ z:=3}(T),\ \{a' \rightarrow a,\ y \rightarrow x\},\ \text{TRUE},\ \text{TRUE})$$

As an example, fusing two query fragments that scan the same table with different projections:

```
SELECT i_brand_id + 1 AS brand_plus_one
FROM item

SELECT new_brand_id + 1 AS x, 'new brand' AS y
FROM (SELECT i_brand_id AS new_brand_id
      FROM item) t
```

results in (P, M, TRUE, TRUE) where:

```
P: SELECT i_brand_id + 1 AS brand_plus_one,
          'new brand plus one' AS y
   FROM item
M: x → brand_plus_one
```

D. Joins

Consider now J1 = JL1⋈C1JR1 and J2 = JL2⋈C2JR2. For simplicity we assume inner joins, but the results are similar for outer- and semi-join variants. To fuse J1 and J2 we first attempt to pairwise fuse the left and right sides of J1 and J2, and return ⊥ if any recursive call fails to deliver a fused result. Otherwise, assume Fuse(JL1, JL2) = (JL, ML, LL, RL) and Fuse(JR1, JR2) = (JR, MR, LR, RR). We first obtain the resulting column mapping as M = ML∪MR since ML and MR are non-overlapping. Using this mapping, we check whether the join conditions in J1 and J2 are equivalent, that is, whether C1 ≡ M(C2). If this is not the case, we return ⊥[3]. Otherwise, we define Fuse(J1, J2) as:

$$(JL \bowtie_{C1} JR,\ M,\ LL \text{ AND } M(LR),\ RL \text{ AND } M(RR))$$

For instance, fusing two fragments that join the same tables:

```
SELECT ss_store_sk AS a_sk, ss_quantity
FROM store_sales JOIN item
ON ss_item_sk = i_item_sk
WHERE ss_addr_sk > 20 AND i_size IN ('m', 'l')

SELECT ss_store_sk AS b_sk
FROM store_sales, item
ON ss_item_sk = i_item_sk
WHERE i_size = 'l'
```

results in (P, M, L, R) where

```
P: SELECT ss_store_sk AS a_sk, ss_quantity
   FROM store_sales JOIN item
   ON ss_item_sk = i_item_sk
   WHERE i_size = 'l'
     OR (ss_addr_sk > 20 AND i_size IN ('m', 'l'))
M: b_sk → a_sk
L: ss_addr_sk > 20 AND i_size IN ('m', 'l')
R: i_size = 'l'
```

Note that this approach would not fuse trees with different join orders. Although in principle this sounds rather limiting, in practice (i) many queries use common table expressions (CTEs) to reuse computation and thus start with the same join tree, and (ii) an earlier compilation phase of join canonicalization typically produces compatible join orders for subqueries that can be fused. A more general handling of joins, however, is part of future work and requires flattening the join tree into an n-ary join and performing a more sophisticated matching of its inputs via techniques inspired from materialized view matching.

E. Aggregations

Aggregate functions in Athena have additional syntactic sugar that simplifies the treatment of fusion. Specifically, each aggregate is a pair (a, m) where a is a traditional aggregate function (e.g., COUNT(*)) and m is a boolean expression, called the *mask* of the aggregate. The semantics of this construct is that during aggregation we only consider input tuples that satisfy the mask for the purpose of the aggregate, and discard the rest. Note that each aggregate function in a GroupBy operator can have different mask expressions, and thus GroupBy operators can compute aggregations over different subsets of data. Consider $G1=\text{GroupBy}_{K1,A1}(P1)$, where K1 is the set of grouping columns and A1 is the list of aggregate functions $[c_i:=(a_i,m_i)]$, and analogously consider $G2=\text{GroupBy}_{K2,A2}(P2)$.

To fuse G1 and G2 we proceed as follows. We first attempt to recursively fuse the inputs and propagate any failure ⊥. Otherwise, the fusion of P1 and P2 returns (P, M, L, R). We then check whether the grouping columns are equivalent (i.e., whether K1 = M(K2)) and return ⊥ if they are not. Otherwise, we initialize the new mapping $M_{new} = M$, and assemble the new aggregations A_{new}. To that end, for each $c_i:=(a_i,m_i)$ in A1, we add to A_{new} an aggregation with a tighter mask $c_i:=(a_i,\ m_i$ AND L). Then, for each $c_j:=(a_j,m_j)$ in A2, we check whether $(M(a_j),\ M(m_j$ AND R)) already exists in A_{new}. If it does (and is assigned to some variable c_{new}) we add $c_j \rightarrow c_{new}$ to the new mapping M_{new}. If it does not, we add a new aggregate $c_j := (M(a_j),\ M(m_j$ AND R)) to A_{new} (note that the new column c_j does not exist in A_{new} or P and therefore this is well-formed).

[3] Note that we can still address scenarios in which the conditions do not fully match by calculating the common portion and treat residuals as filters.

There is a subtle detail before assembling the final result. When we deal with non-scalar GroupBy operators (i.e., non-empty grouping columns), aggregations with masks return an aggregated row even if all input rows have been discarded by the mask. This is not the desired result during fusion, since groups for which all input rows were discarded should not produce any row. Thus, for non-scalar GroupBy operators for which L (respectively, R) are not TRUE, we add to A_{new} compensating aggregates countL:=(COUNT(*), L) (respectively, countR:=(COUNT(*), R)), and define compensating filters compL ≡ countL>0 (respectively, compR ≡ countR>0) . For a scalar aggregate or in case L (or R) are TRUE, we do not modify A_{new} and define compL (or compR) as TRUE. With all this preparation, we define Fuse(G1, G2) as:

$$(\text{GroupBy}_{K1,\ A_{new}}(P),\ M_{new},\ compL,\ compR)$$

As an example, given G1 and G2 defined as:

$$G1 = \text{GroupBy}_{\{a\},\ x:=(SUM(b),\ TRUE)}(\text{Filter}_{c=1}(T))$$
$$G2 = \text{GroupBy}_{\{a\},\ y:=(AVG(b),\ d=1)}(T))$$

The result of fusing G1 and G2 is (P, M, z>0, TRUE), where M is the trivial mapping from columns in T, and P is

$$GB_{\{a\},[x:=(SUM(b),c=1),y:=(AVG(b),d=1),z:=(COUNT(*),c=1)]}(T)$$

As another example, fusing two query fragments that aggregate on the same column of the same table:

```
SELECT i_item_sk,
       MIN(i_brand_id) AS mi
FROM item
WHERE i_color = 'red'
GROUP BY i_item_sk

SELECT i_item_sk,
       AVG(i_category_id)
         FILTER (WHERE i_size = 'm') AS avgc
FROM item
GROUP BY i_item_sk
```

results in (P, ∅, TRUE, TRUE) where:

```
P: SELECT i_item_sk,
          MIN(i_brand_id)
            FILTER (WHERE i_color = 'red') AS mi,
          AVG(i_category_id)
            FILTER (WHERE i_size = 'm') AS avgc
FROM item
GROUP BY i_item_sk
```

An important consequence of implementing query fusion without creating new custom operators, is that orthogonal rules are applicable to fused results (e.g., expression simplification over masks, or rules over GroupBy operators can automatically operate over intermediate results with no changes).

F. Distinct Aggregations

In addition to traditional mechanisms to implement distinct aggregates (e.g., via self-joins), Athena offers an alternative that relies on a new relational operator called MarkDistinct. This operator is defined by a relational input P and a set D of columns in P and returns a new Boolean column d (we denote this as $\text{MarkDistinct}_{d\leftarrow D}(P)$). This operator passes through the input P and assigns new values to column d. The value of d is TRUE each time a combination of values of D is seen for the first time in P, and FALSE otherwise. In conjunction with masks, we can implement distinct aggregates using the MarkDistinct operator. For instance, the expression $\text{GroupBy}_{\{a\},x:=count(distinct\ b),\ y:=count(distinct\ c)}(T)$ can be implemented as:

$$\text{GroupBy}\{a\},[x:=(count(b),db),y:=(count(c),dc)]$$
$$\text{MarkDistinct}_{db\leftarrow\{b\}}$$
$$\text{MarkDistinct}_{dc\leftarrow\{c\}}$$
$$T$$

Consider now fusing $M1=\text{MarkDistinct}_{d1\leftarrow D1}(P1)$ and $M2=\text{MarkDistinct}_{d2\leftarrow D2}(P2)$. If Fuse(P1, P2) returns ⊥, we propagate this value. Otherwise, it returns (P, M, L, R), and we define Fuse(M1, M2) as (Q, M, L, R), where

$$Q \equiv \text{MarkDistinct}_{d1\leftarrow D1\cup\{m1\}}$$
$$\text{MarkDistinct}_{d2\leftarrow D2\cup\{m2\}}$$
$$\text{Project}_{m1:=L,\ m2:=R}$$
$$P$$

That is, we define new boolean columns m1 and m2 for the compensating filters L and R, and add those new columns to the corresponding MarkDistinct column sets to operate on. This way, MarkDistinct operators would distinguish the first time they process a new instance value of the original column set and whether this value satisfied or not the compensating filters.

There are a few optimizations to this basic scheme, like extending the MarkDistinct operator itself to consider masks natively, skipping the generation of extra columns from the Project operator if the compensating filters are TRUE, or processing a chain of MarkDistinct operators on both sides holistically rather than one pair at a time. We omit those extensions to simplify the presentation.

G. Additional details

So far, we showed how we define the Fuse operation whenever both inputs have the same root operator, and for a selected set of relational operators. We next describe some additional details that extend the mechanism described so far to handle more scenarios.

First, there are some operators that accept a default implementation of the Fuse operation. Consider for instance the EnforceSingleRow operator, which takes an input and enforces that it returns a single row (or otherwise fails the query). In this case, the fusion of two queries that have this operator as the root can be done in a generic way, by (i)

recursively applying the Fuse operation to the input of the root operator, (ii) checking whether the operators are equivalent modulo the returned mapping, and (iii) replacing the child of the first query with the result of the recursive Fuse call.

Second, we outline a best-effort approach to generalize fusion whenever the root operators of the input queries are not the same. Suppose that the left operator is MarkDistinct and the right operator is not. Because MarkDistinct just adds a new column to the result, we can (i) skip the MarkDistinct operator on the left side, (ii) fuse its child with the right side, and (iii) if successful, add back the MarkDistinct operator to the fused result. Other extensions include manufacturing a trivial projection if one of the queries has a Project root operator and the other does not, or add a trivial filter (TRUE) if the root of one subquery has a Filter operator and the other does not. These extensions are subtle since they can produce correct but inefficient fused queries. For instance, suppose that P1 = Filter(T) and P2 = MarkDistinct(Filter(T)). In this case, we could either skip the MarkDistinct operator in P2 and match the child with P1, or manufacture a trivial filter on top of P2 and fuse it with P1. Clearly, the first alternative is better and would produce a good fused result, while the second alternative would fail to push the disjunction of both filters down to the scan due to the injected trivial filter. As this example demonstrates, a careful mechanism to decide compensating actions in case of non-matching root operators is crucial to have a flexible and performant solution. In general, however, more fundamental extensions to generalize fusion for queries that are expressed in structurally different ways is an important research topic and part of future work.

IV. Optimization Rules

We now describe specific optimization rules we incorporated into the Athena engine, which use query fusion as a building block. We first describe individual rules in a simplified way, and later complement the presentation with some additional details that make them more widely applicable.

A. GroupByJoinToWindow

This rule transforms a common pattern in which a common expression is aggregated and joined back to itself to obtain additional information on the aggregated rows. Intuitively it is a calculation that extends an input relation with aggregates computed on a subset of columns. Window functions operate in this manner and can be used to rewrite the original pattern.

Consider an input pattern like P1 $\bowtie_C$ GroupBy$_{K,A}$(P2) and suppose that the following conditions hold:

- Fuse(P1, P2) = (P, M, TRUE, TRUE)[4].
- The join condition C can be written as C1 AND C2, where C1 = cl_1=cr_1 AND ... AND cl_n=cr_n, and moreover, K = {cr_1,...,cr_n}. In other words, the join condition contains equalities with columns that match the grouping columns on the right-hand side and remaining conditions C2 that would result in residuals.
- For each cl_i=cr_i in C1, we have that cl_i = M(cr_i), so that the join can be seen as an equijoin (with extra predicates) modulo the mapping function M.

In this case, we replace the original pattern with:

```
Filter M(C2)
  Window A OVER(PARTITION BY cl1...cln)
    Filter AND_{i=1..n} (cli IS NOT NULL)
      P
```

The input pattern can either be directly specified in the input query or via a decorrelation opportunity [20] for queries like:

```
SELECT * FROM T T1
WHERE T1.a = (SELECT AVG(c)
              FROM T T2
              WHERE T2.b = T1.b)
```

B. JoinOnKeys

This rule addresses a common pattern in which similar subqueries, which return different views of the same data, are *self-joined* together. Because of the existence of keys, each row from the left matches with at most one row from the right. Therefore, we are extending each row that matches with columns from both sides. A NOT NULL condition is needed to remove rows that do not have a match on the other side, and residual predicates are added to ensure the original result.

Consider a pattern like P1 $\bowtie_C$ P2 and suppose the following conditions hold:

- Fuse(P1, P2) = (P, M, L, R).
- A key of P1 is K1 (i.e., every row from P1 can be uniquely identify by the values of columns in K1), and, analogously, a key of P2 is K2.
- The join condition C can be written as C1 AND C2, with C1 ≡ cl_1=cr_1 AND ... AND cl_n=cr_n. Moreover, it holds that K1 = {cl_1,...,cl_n} and K2 = {M(cr_1),...,M(cr_n)}. In other words, the join condition contains equalities on columns that match the keys of the left and right sides of the join.

In this case, we replace the original pattern with the alternative Filter$_{newC}$(P), where:

```
newC ≡ L AND R AND M(C2)
       AND cl1 IS NOT NULL AND ...
       AND cln IS NOT NULL
```

While this is a general rule that depends on the existence of appropriate keys, Athena currently does not have a general mechanism to propagate key information through query plans. For that reason, we implemented specific variations of this rule

[4] To simplify the presentation, we consider the case in which the filter conditions of the fused result are TRUE. The extension to arbitrary conditions is more involved, and requires using masks, and compensating aggregations and filters, in a similar way to the fusion of non-scalar aggregations.

to scenarios that we can guarantee correctness. For instance, we added a rule to handle the case of $P1 = GroupBy_{K1, A1}(Q1)$ and $P2 = GroupBy_{K2, A2}(Q2)$ (leveraging the fact that K1 and K2 are the corresponding keys). An interesting special case of this rule happens when both K1 and K2 are empty (e.g., for scalar aggregates) and the subsequent join is a cross product. In this case, given $GroupBy_{\emptyset,A1}(Q1) \times GroupBy_{\emptyset,A2}(Q2)$ and Fuse(Q1, Q2) = (Q, M, L, R), we transform the input pattern into $Filter_{L \text{ AND } R}(GroupBy_{\emptyset,A1 \cup M(A2)}(Q))$.

C. UnionAllOnJoin

This rule handles scenarios in which customers combine results of two computations that are very similar overall, but differ on a single table (e.g., they union together some analytical insight applied over different fact tables). To simplify the exposition, we consider the case of semi-join operators as the root of the plans that are fused, but the same ideas can be extended to other join types.

Consider a pattern $UnionAll(P1 \ltimes_{c1} Z1, P2 \ltimes_{c2} Z2)$ and suppose the following conditions hold:

- Fuse(Z1, Z2) = (Z, M, L, R).
- The semi-join conditions are composed of column equalities ($c1_i = d1_i$ for C1 and $c2_i = d2_i$ for C2), and we can match the right-hand sides of those conjuncts modulo mappings (i.e., $d1_i = M(d2_i)$).

We can let the union operation happen first between the right tables, and have the result join back to the common left side. That is, we replace the original pattern with:

$SemiJoin_{UM(C1) \text{ AND } ((tag=1 \text{ AND } UM(L)) \text{ OR } (tag=2 \text{ AND } UM(R)))}$
  UnionAll
    $Project_{tag:=1,UA1}(P1)$
    $Project_{tag:=2,UA2}(P2)$
  Z

That is, we push the UnionAll operation below the semi-join, and tag each input with a unique number that is then used to reconstruct the semi-join predicate. A subtle detail is sketched above by the UM/UA1/UA2 constructs. Recall that a UnionAll operator takes two input schemas and positionally maps them to its own output schema (we call such mapping UM). Since we pushed the UnionAll below the two original semi-joins, we somehow unified their left input schemas. This is fine for the overall result, since the positional mapping of the original UnionAll is preserved. However, the condition of the semi-join itself might use columns that are not propagated correctly and must be handled appropriately. Consider a conjunct of C1, say, $c1_i = d1_i$ and its corresponding conjunct from C2, $c2_i = d2_i$. By definition, we know that $d1_i = M(d2_i)$. We additionally require that the left-hand side of the resulting equality is well formed. Specifically, we need that there is a column in the output of the pushed UnionAll that returns $c1_i$ for rows coming from P1, and $M(c2_i)$ for rows coming from P2. This column, in general, might not be part of UM in the original UnionAll operator. If that case, we project a new column on each input of the new UnionAll with the appropriate value. We represent this information as an extra assignments UA1 and UA2 from columns of P1 and P2 used in the semi-join predicate, and add those columns to the UM mapping. The extended mapping UM is used to convert C1, L, and R from input to output columns in the resulting UnionAll. As a simple example (using SQL for simplicity), a query like:

```
SELECT P1.a FROM P1
WHERE P1.b IN (SELECT z1 FROM Z)
UNION ALL
SELECT P2.c FROM P2
WHERE P2.d IN (SELECT z1 FROM Z)
```

would be transformed as follows (note the new columns b and d in the new UNION ALL input schemas):

```
SELECT newA FROM
  (SELECT a AS newA, b as newB FROM P1
   UNION ALL
   SELECT c AS newA, d AS newB FROM P2
   ) innerT
WHERE newB IN (SELECT z1 FROM Z)
```

Note that this rule can be recursively applied to the result (e.g., if P1 and P2 themselves are joins). In general, we can extend this rule to deal with n-ary UnionAll operators, but we omit those details to simplify the presentation.

D. UnionAll

Another common pattern that customers use is to compute a common expression and then union non-necessarily disjoint subsets of the result with different projections, as shown below:

```
WITH cte AS (...)
SELECT a, b FROM cte WHERE p1
UNION ALL
SELECT a, c FROM cte WHERE p2
```

In this case, we only need to read the source table once, but we can generate as many replicas as necessary, depending on how many union branches there are. We then extract the corresponding output from each replica per each of the original union branch filters and projections. A cross-join between the source table and an artificial constant table can be used for this purpose. Specifically, consider a pattern like UnionAll(P1, P2) and suppose that Fuse(P1, P2) = (P, M, L, R) and UM is the positional mapping of columns $c1_i$ from P1 and $c2_i$ from P2 into the output columns of the UnionAll operator. In that case we transform the input pattern into:

```
Project UM(c1i):=CASE WHEN tag=1 THEN c1i ELSE M(c2i) END, ...
  Filter (tag=1 AND L) OR (tag=2 AND R)
    CrossJoin
      P
      ConstantTable((1),(2)) as Temp(tag)
```

Specifically, we cross-join the fused expression P with a constant table with two rows (with values 1 and 2). We then filter the combined result distinguishing the compensating

filters L and R with the tag that represents the lineage of each row, and produce the right output columns by selecting the corresponding input columns for each output of the UnionAll via a CASE statement.

Some extensions to this basic scheme include:

- Handling n-ary UnionAll operators by extending the constant table definition, filters and projections.
- Removing the CASE statements if $c1_i = M(c2_i)$ and replacing them with $UM(c_1 i) := c_1 i$.
- If we detect a contradiction in the compensating fused filters (i.e., L AND R ≡ FALSE), replacing the overall transformation with a simpler alternative:

```
Project UM(c1i):=CASE WHEN L THEN c1i ELSE M(c2i) END, ...
  Filter L OR R
    P
```

E. Additional details

We now explore extensions to the rules we introduced, which result in additional coverage on common scenarios.

First, the rules as described so far sometimes are strict with respect to the patterns they require. For instance, the pattern that triggers the *GroupByJoinToWindow* rule in Section IV-A is $P1 \bowtie_C GroupBy_{K,A}(P2)$. In general, there could be a filter pushed in between the join and the group-by operator (e.g., a single-column predicate on an aggregate column in A). Or there could be a Project operator in between the Join and GroupBy, generating an expression that is used as a residual condition in C. The extensions to handle these scenarios leverage relational algebra equivalences (e.g., pulling a filter above a join, or carrying over projections across our transformations). Note that these extensions need to be handled with care (e.g., we can pull a condition above an inner join, but not in general for outer variants).

Second, it is important to gracefully handle n-ary operators like unions and joins. There are two approaches that we follow. The first one is to rely on the rule engine itself and the fact that fused subplans are still defined in terms of existing relational algebra operators. Therefore, handling n-ary operators can be done by a sequence of pairwise invocations of the rule. This approach works well for scenarios in which the result of the rule is similar enough to the original fused inputs that a subsequent iteration is possible. An example of this scenario is the *JoinOnKeys* rule (Section IV.B), for which we can linearize the input tree by flattening joins and incrementally grow the fused result two inputs at a time. Another approach is to extend the Fuse operation to work natively on more than two inputs and generalize the rules accordingly. This works well for the *UnionAll* rule (Section IV.D) since applying a sequence of rules to binary fragments of an n-ary UnionAll results in unnecessarily complex intermediate results.

Third, an interesting problem is the interaction between the rules in this section and other rules present in the engine. For instance, Athena performs join reordering, and in fact, the specific order of inputs in a join (including topology like left-deep vs. bushy trees) influences whether rules based on query fusion can be applied. Implementing the rules in this section so that they operate over every join order is a possibility, but it quickly becomes prohibitively expensive. Instead, we extend join-based rules (Sections IV.A and IV.B) so that they operate before join reordering. Specifically, after they match a root join operator, we (i) recursively traverse its inputs to conceptually obtain an n-ary join, and (ii) attempt to apply rules pairwise to specific join inputs (and intermediate rule results) a quadratic number of times.

Finally, there is the problem of cost-based applicability of the rules. In general, rules operate at the logical level and therefore it is not possible to compare costs of the plans before and after rule applications. Optimization frameworks like Cascades [3] allow deep exploration of alternatives and naturally solve this problem by delaying the decision of choosing the best plan after all alternatives have been fully optimized. Athena's optimizer does not yet support this form of exploration, so we rely on local heuristics based on statistics and plan properties to decide the applicability of each rule.

V. Experimental Results

We now report an experimental evaluation of our techniques over the TPC-DS benchmark [16]. We briefly show general quantitative results on the performance of our approach, and then examine in detail specific queries to showcase how our techniques operate. For our experiments, we used a TPC-DS dataset at scale factor of 3TB (which is consistent with the 99-th percentile of data sizes that Athena customers currently operate on), and the corresponding queries described in the benchmark. We partitioned the largest 7 tables (store_sales, store_returns, catalog_sales, catalog_returns, web_sales, web_returns, and inventory) by appropriate date columns, which results in layouts with 200 to 2000 partitions. The remaining tables were stored without partitioning. All tables were encoded using Parquet with Snappy compression. We evaluated all queries using Athena's default production configuration which serves normal customer queries. Athena allocates resources to queries depending on various signals including data volumes, query complexity, and historical information. In our experiments, we let Athena choose appropriate resources for each query, but used the same configuration for both the baseline (i.e., without our optimizations) and an instrumented version of the compiler (i.e., including our new optimizations).

We observed that our approach improves the overall execution time of the 99-query workload by 14% compared to the baseline. However, not all queries trigger our optimization rules. When restricted to those that changed plans, we observed a 60% improvement in performance on average, with some queries improving performance over 6 times. The rest of the queries either did not change plans when incorporating our optimization rules, or their performance characteristics remained largely unchanged. This shows that our techniques are not applicable to all possible queries but instead target specific patterns. However, those patterns are not uncommon in complex analytical queries, especially in those with common table subexpressions. This observation is backed by analysis on real customer data.

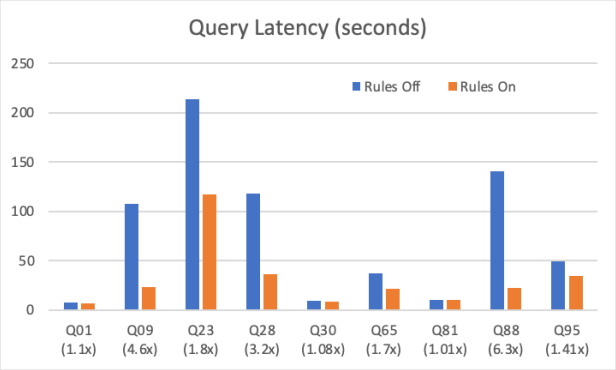

Figure 1: Latency improvement for selected queries.

Figure 1 shows the performance improvement of our techniques over select queries in the benchmark. We see that while some queries result in moderate improvements (below 10%), some others can be over 6 times faster when the new optimizations take place.

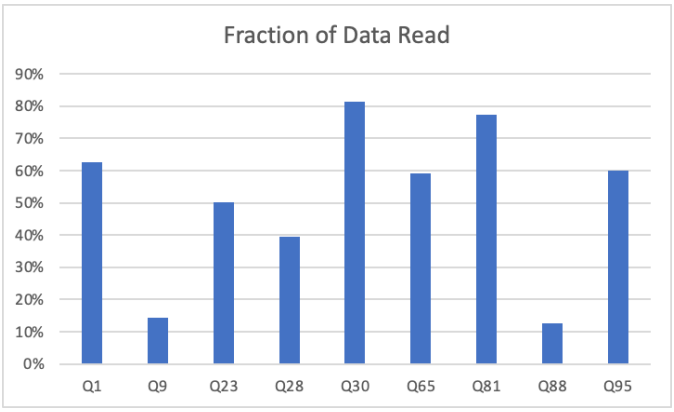

Figure 2: Reduction in data read for selected queries.

Figure 2, in turn, shows the fraction of input data that each query read from S3 using our approach, compared to the baseline. The figure shows that the reduction in I/O can be drastic, with some queries reading just 15% of the amount of data compared to the baseline. In general, all queries in the figure result in at least ~20% reduction in scanned data which, due to the pay-as-you-go model that Athena uses, directly translates to customer savings.

In the rest of this Section, we dive deeper into these selected queries to showcase how our proposed approach affects the resulting execution plans. We use bold text to emphasize the query fragments that change after our optimizations.

A. *Introducing Window Operators*

Queries Q01, Q30 and Q65 include variants of the fragment we used in Section I to motivate our approach. We focus on Q1 next since it looks, at first sight, different from the motivating example. Specifically, Q01 is:

```
WITH customer_total_return AS (
  SELECT sr_customer_sk AS ctr_customer_sk,
         sr_store_sk AS ctr_store_sk,
         sum(sr_return_amt) AS ctr_total_return
  FROM store_returns, date_dim
  WHERE sr_returned_date_sk = d_date_sk
    AND d_year = 2000
  GROUP BY sr_customer_sk, sr_store_sk)
SELECT c_customer_id
FROM customer_total_return ctr1, store, customer
WHERE ctr1.ctr_total_return > (
  SELECT avg(ctr_total_return)*1.2
  FROM customer_total_return ctr2
  WHERE ctr1.ctr_store_sk = ctr2.ctr_store_sk)
AND s_store_sk = ctr1.ctr_store_sk
AND s_state = 'TN'
AND ctr1.ctr_customer_sk = c_customer_sk
ORDER BY c_customer_id LIMIT 100
```

In this case, the query can be decorrelated, which results in a pattern that triggers the *GroupByJoinToWindow* rule. We replace the join of `ctr1` with the aggregated version coming out of the decorrelation over `ctr2`. Note that the two inputs that are needed for the rule are not next to each other, but separated by other joins (on `store` and `customer`). Our extensions to handle n-ary joins, described in Section IV.E, produce an alternative that removes the common expression and replaces it with a single `WINDOW` operator. The resulting rewrite, given below in SQL for simplicity, is then:

```
WITH customer_total_return AS (
  SELECT sr_customer_sk AS ctr_customer_sk,
         sr_store_sk AS ctr_store_sk,
         sum(sr_return_amt) AS ctr_total_return
  FROM store_returns, date_dim
  WHERE sr_returned_date_sk = d_date_sk
    AND d_year = 2000
  GROUP BY sr_customer_sk, sr_store_sk)
SELECT c_customer_id
FROM store,
     customer,
     (SELECT *,
             1.2 * AVG(ctr_total_Return) OVER
               (PARTITION BY ctr_store_sk) AS aCtr
      FROM customer_total_return) ctr
WHERE ctr.ctr_total_return > ctr.aCtr
  AND s_store_sk = ctr.ctr_store_sk
  AND s_state = 'TN'
  AND ctr.ctr_customer_sk = c_customer_sk
ORDER BY c_customer_id LIMIT 100
```

Queries rewritten in this way result in modest improvements in latency (below 10%, as the data scans are in parallel) but read 20% to 40% less data, which directly translates in cost savings for customers. While evaluating these results, we identified some opportunities to improve the performance of `Window` operators, which would directly translate in better performance overall. Additionally, we noticed that these queries use less CPU (with savings ranging from 20% to 40%). This implies that we could further improve our cluster utilization by reducing the nodes allocated to these queries, which is an active direction we are pursuing.

B. Merging Scalar Aggregates

Queries Q09, Q28 and Q88 combine aggregates over the same common expression using slightly different predicates. We focus on Q09 as a representative of these queries, given by the fragment below:

```
SELECT CASE
  WHEN (SELECT COUNT(*)
        FROM store_sales
        WHERE ss_quantity BETWEEN 1 AND 20) > 48409437
  THEN (SELECT AVG(ss_ext_discount_amt)
        FROM store_sales
        WHERE ss_quantity BETWEEN 1 AND 20)
  ELSE (SELECT AVG(ss_net_profit)
        FROM store_sales
        WHERE ss_quantity BETWEEN 1 AND 20) END
  AS bucket1,
  <4 more variations of the CASE expression above>
FROM reason
WHERE r_reason_sk = 1
```

In total, there are 15 scans of the fact table `store_sales`, each one returning various aggregates over different subsets of data. The engine first performs subquery removal and transforms the various expressions in the `CASE` statements into relational subtrees connected via cross products, and then the *JoinOnKeys* rule, specialized to handle scalar aggregates and cross joins, is triggered. The resulting plan for Q09, described below using SQL for clarity, is given by:

```
SELECT CASE WHEN v1 > 48409437 THEN t1 ELSE e1 END,
       <4 more variations>
FROM (
 SELECT COUNT(*) FILTER(WHERE b1) AS v1,
        AVG(ss_quantity) FILTER(WHERE b1) AS t1,
        AVG(ss_net_profit) FILTER(WHERE b1) AS e1,
        <4 more variations>
 FROM (
    SELECT *, ss_quantity between 1 and 20 as b1,
              <4 more variations>
    FROM store_sales
    WHERE ss_quantity between 1 and 20
       OR <4 more variations>)),
 Reason
 Where r_reason_sk = 1)
```

In general, this pattern results in the largest improvements both in latency (from 3x to 6x improvements) and bytes read from S3 (from 60% to 85% reduction in scanned bytes and, consequently, cost). While the transformations are similar in these 3 queries, there are some minor differences. Q88 has a complex common expression involving a 4-way join, which results in even more savings in latency. Q28 uses distinct aggregates, which leverages our extensions to deal with `MarkDistinct` operators during query fusion.

C. Refactoring UnionAll Branches

A simplified version of query Q23, which combines two similar insights over different fact tables is as follows:

```
WITH freq_items AS (...), best_customer AS (...)
SELECT SUM(sales)
FROM (SELECT cs_quantity*cs_list_price AS sales
      FROM catalog_sales, date_dim
      WHERE d_year = 1999
        AND d_moy = 1
        AND cs_sold_date_sk = d_date_sk
        AND cs_item_sk IN
              (SELECT item_sk FROM freq_items)
        AND cs_bill_customer_sk IN
              (SELECT cust_sk FROM best_customer)
      UNION ALL
      SELECT ws_quantity*ws_list_price AS sales
      FROM web_sales, date_dim
      WHERE d_year = 1999
        AND d_moy = 1
        AND ws_sold_date_sk = d_date_sk
        AND ws_item_sk IN
              (SELECT item_sk FROM freq_items)
        AND ws_bill_customer_sk IN
              (SELECT cust_sk FROM best_customer))
```

This query uses a `UnionAll` operation to combine two fragments that have almost the same structure except for the fact tables being used (`catalog_sales` vs. `web_sales`):

`web_sales` $\bowtie$ `date_dim` $\ltimes$ `freq_items` $\ltimes$ `best_customer`, and
`catalog_sales` $\bowtie$ `date_dim` $\ltimes$ `freq_items` $\ltimes$ `best_customer`

Rule *UnionAllOnJoin* triggers repeatedly in this case, first fusing `best_customer`, then `freq_items`, and finally `date_dim`. The resulting plan (using `SQL`) is as follows:

```
WITH freq_items as (...), best_customer as (...)
SELECT SUM(sales) FROM (
  SELECT cs_quantity*cs_list_price AS sales
  FROM date_dim, (
    SELECT cs_sold_date_sk, cs_quantity, cs_item_sk,
           cs_bill_customer_sk, cs_list_price
    FROM catalog_sales
    UNION ALL
    SELECT ws_sold_date_sk, ws_quantity, ws_item_sk,
           ws_bill_customer_sk, ws_list_price
   FROM web_sales)
WHERE d_year = 1999
  AND d_moy = 1
  AND cs_sold_date_sk = d_date_sk
  AND cs_item_sk in (SELECT item_sk FROM freq_items)
  AND cs_bill_customer_sk IN
       (SELECT c_customer_sk FROM best_customer))
```

In this case, query latency is almost 2x better, and the bytes scanned (and corresponding costs) are dropped almost by half. The reason is that both `freq_items` and `best_customer` are rather expensive common expressions, which make a big difference when one instance is removed. Another benefit of this rewrite concerns the amount of memory that the query uses. At larger scale factors, we noticed that the engine runs out of working memory and starts spilling to disk intermediate state encoded in join and aggregate hash tables. One reason is that both instances of the common subexpressions are evaluated concurrently. When removing the common expressions, the amount of memory needed to hold intermediate state is reduced by half as well, and spilling is not needed. We have seen an additional 50% improvement in latency for those scenarios.

D. Unifying Relational Aggregates

Query Q95 exhibits a curious pattern. A simpler version is:

```
WITH ws_wh as (
  SELECT ws1.ws_order_number as ws_wh_number
  FROM web_sales ws1, web_sales ws2
  WHERE ws1.ws_order_number = ws2.ws_order_number
   AND ws1.ws_warehouse_sk <> ws2.ws_warehouse_sk)
SELECT <scalar aggs>
FROM web_sales,
     date_dim,
     customer_address,
     web_site
WHERE <filter and join predicates>
  AND ws_order_number IN
       (SELECT ws_wh_number FROM ws_wh)
  AND ws_order_number IN
       (SELECT wr_order_number FROM ws_wh
        JOIN web_returns
        ON wr_order_number = ws_wh_number)
```

We can see that the two `IN` clauses are very related to each other. Specifically, the first one is redundant. The reason is that every `ws_order_number` that appears in the second subquery must also appear on the first one (since the second one further restricts the values it returns due to joining with `web_returns` on the same column). Interestingly enough, both subqueries refer to an expensive common expression `ws_wh` that self joins a fact table. Our approach is able to simplify this query relying on the interplay between our new rules and existing ones in the engine. Specifically, we first transform the semi-joins into equivalent joins over a distinct on the right side. Then, we apply a rule that pushes a distinct operation below a join whenever the distinct and join columns agree. In this way, we obtain an alternative that can be expressed in SQL (for simplicity) as:

```
WITH ws_wh as (...)
SELECT <scalar aggs> FROM
 web_sales ws1,
 date_dim,
 customer_address,
 web_site,
 (SELECT DISTINCT ws_wh_number FROM ws_wh) R0,
 (SELECT DISTINCT wr_order_number FROM web_returns) R1,
 (SELECT DISTINCT ws_wh_number FROM ws_wh) R2
WHERE <filter and join predicates>
  AND ws1.ws_order_number = R0.ws_order_number
  AND ws1.ws_order_number = R1.wr_order_number
  AND ws1.ws_order_number = R2.ws_order_number
```

Finally, the *JoinOnKeys* rule triggers and fuses `R0` and `R2`. Since these subqueries (`R0` and `R2`) do not return any aggregate and the grouping columns are the same, the fusion essentially removes one of the duplicate expressions. The resulting query is 30% faster and reads 40% less data from S3.

VI. Conclusion

In this paper we introduced new optimizations implemented in Amazon Athena's query engine that improve performance of a class of queries containing common subexpressions. Our optimizations are built on top of the concept of query fusion, which we extended and streamlined for the purposes of this work. We showed several real-world examples that can benefit from this approach, and quantified the benefit of our query rewrites on the TPC-DS benchmark. We continue identifying additional rewrite rules that happen frequently in customer workloads, and we plan to use in the future the fusion infrastructure of Section III as a building block of our work on generic spooling of subqueries that go beyond the techniques introduced in this paper.

References

[1] Jyoti Leeka and Kaushik Rajan. Incorporating superoperators in big-data query optimizers. In PVLDB, 13(3):348–361, 2019.

[2] Partho Sarthi et al. Generalized Sub-Query Fusion for Eliminating Redundant I/O from Big-Data Queries. 14th USENIX Symposium on Operating Systems Design and Implementation, 2020.

[3] G. Graefe. The Cascades framework for query optimization. Data Engineering Bulletin, 18(3), 1995.

[4] Amazon Athena. https://aws.amazon.com/athena

[5] Amazon Simple Storage Service. https://aws.amazon.com/s3

[6] The JavaScript Object Notation (JSON) Data Interchange Format. https://datatracker.ietf.org/doc/html/rfc8259

[7] The ORC Specification Format. https://orc.apache.org/specification/ORCv1

[8] Apache Avro Specification. https://avro.apache.org/docs/current/spec.html

[9] Apache Parquet Specification. https://parquet.apache.org/documentation/latest/

[10] Snappy, a fast compressor/decompressor. https://github.com/google/snappy/

[11] A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://www.zlib.net/

[12] GZip file format specification version 4.3. https://datatracker.ietf.org/doc/html/rfc1952

[13] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, Christopher Berner. Presto: SQL on Everything. In Proceedings of IEEE ICDE, 2019.

[14] AWS Glue Documentation. https://docs.aws.amazon.com/glue

[15] AWS Lambda Documentation. https://docs.aws.amazon.com/lambda

[16] Transaction Processing Performance Council. TPC Benchmark DS. http://tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf

[17] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational data processing in spark. In Proceedings of the 2015 ACM SIGMOD, 2015.

[18] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, Darren Shakib. SCOPE: parallel databases meet MapReduce. VLDB J. 21(5): 611-636 (2012)

[19] Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive—a warehousing solution over a MapReduce framework. In: Proceedings of VLDB Conference (2009)

[20] Cesar Galindo-Legaria, Milind Joshi. Orthogonal optimization of subqueries and aggregation. In Proceedings of Sigmod (2001).

[21] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In Proceedings of Sigmod (2007).

[22] Amazon Redshift. https://aws.amazon.com/redshift

[23] Amazon EMR. https://aws.amazon.com