Generalized Sub-Query Fusion for Eliminating Redundant I/O from Big-Data Queries

Partho Sarthi, Kaushik Rajan, and Akash Lal, Microsoft Research India; Abhishek Modi, Prakhar Jain, Mo Liu, and Ashit Gosalia, Microsoft; Saurabh Kalikar, Intel

https://www.usenix.org/conference/osdi20/presentation/sarthi

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX
Abstract

SQL is the de-facto language for big-data analytics. Despite the cost of distributed SQL execution being dominated by disk and network I/O, we find that state-of-the-art optimizers produce plans that are not I/O optimal. For a significant fraction of queries (25% of popular benchmarks like TPCDS), a large amount of data is shuffled redundantly between different pairs of stages. The fundamental reason for this limitation is that optimizers do not have the right set of primitives to perform reasoning at the map-reduce level that can potentially identify and eliminate the redundant I/O.

This paper proposes RESIN, an optimizer extension that adds first-class support for map-reduce reasoning. RESIN uses a novel technique called Generalized Sub-Query Fusion that identifies sub-queries computing on overlapping data, and fuses them into the same map-reduce stages. The analysis is general; it does not require that the sub-queries be syntactically the same, nor are they required to produce the same output. Sub-query fusion allows RESIN to sometimes also eliminate expensive binary operations like Joins and Unions altogether for further gains.

We have integrated RESIN into SPARKSQL and evaluated it on TPCDS, a standard analytics benchmark suite. Our results demonstrate that the proposed optimizations apply to 40% of the queries and speed up a large fraction of them by $1.1 - 6\times$, reducing the overall execution time of the benchmark suite by 12%.

1 Introduction

SQL is the de-facto language for performing big-data analytics. As there are many alternative ways to express the same query in SQL, query optimizers employ SQL-to-SQL rewrite rules to find equivalent queries that are likely to run faster. The rewritten query is compiled down to an executable plan that consists of many map or reduce stages. Each stage in the plan then runs in a data-parallel manner on many machines.
tions that produce a single map-reduce program. The logical rewrite rules in BLITZ (as in a standard optimizer) only make local transformations. They substitute a connected set of operators with a single super-operator. As a result, BLITZ can only eliminate redundant I/O from specific types of sub-queries with self-joins or self-union. This turned out to be insufficient on a standard benchmark suite like TPCDS where BLITZ only applies to a small fraction (2%) of queries.

This paper introduces RESIN, an optimizer extension that eliminates redundant I/O from complex multi-stage multi-input queries. This fundamentally requires new techniques. As shown in Figure 1, RESIN performs map-reduce reasoning right from the beginning. It introduces two generic logical operators, a parameterized mapper (RESINMAP) and a parameterized reducer (RESINREDUCE) that are each capable of implementing complex sub-queries. RESIN introduces new rules that fuse operators from different parts of the query tree that are processing overlapping sets of data. The fusion relies on the additional expressiveness of RESINMAP and RESINREDUCE. The fusion further enables the elimination of binary operators from the query. Binary operators are particularly expensive as they typically induce multiple shuffles [10]. Compared to BLITZ, we significantly broaden the optimization opportunities: RESIN applies to 38% of TPCDS.

We integrated RESIN with SPARK [5, 26], a popular open-source big-data system, and evaluated on the entire TPCDS suite. Our results demonstrate that RESIN optimizations apply to 40 of the 104 queries in the suite, and speed up 25% of the queries by a significant fraction (average 1.4 ×). RESIN brings down the cumulative execution time for the entire benchmark suite by 12%.

The rest of the paper is organized as follows. Section 2 gives an overview of optimizations performed by RESIN. Section 3 formally defines the query language and introduces RESIN operators. Section 4 describes the core optimizations, sub-query fusion and binary operator elimination. Section 5 presents some key features of our implementation. Section 6 reports our evaluation and Section 7 discusses related work.

2 Overview

This section provides an overview of RESIN. Consider a (fictitious) IoT application that collects readings from multiple sensors deployed all over the world and derives intelligence from it through SQL queries. Each device emits a single message every few hours with two readings corresponding to two different times. The message has the following fields, \(\langle id, hr_1, signal_1, hr_2, signal_2\rangle\) where \(id\) is the device identifier, \(hr_1\) is the hour at which the first reading was taken, \(signal_1\) is the value of the first reading. Similarly, \(hr_2\) and \(signal_2\) are the hour and value of the second reading. The collective log, which can reach Billions of entries across all devices, is processed once a month using SQL queries. We describe RESIN optimizations on two example queries.

Example 1 The query is shown in Figure 2(a)\(^1\). The query separates the subset of columns \(\{id, hr_1, signal_1\}\) and \(\{id, hr_2, signal_2\}\) of each row of the rawLogs table to get intermediate tables V1 and V2, and then performs a Union to put them together. The Union operator performs a multi-set union, i.e., it does not remove duplicate rows from the output. (In general, all our queries operate with multi-set semantics.) Each of V1 and V2 additionally requires a filter to check for the validity of the input (hr fields are in the expected ranges and the signal fields are valid). Figure 2(b) shows a small input table with 5 rows and the result of executing the query on that input. Each of V1 and V2 will contain 4 rows each and the final output signals has 8 rows. For a production-sized execution, imagine scaling each table by a factor of a Billion.

Figure 2(c) shows the execution plan for this query generated by SPARK. The plan employs duplicate scan operators, thus, it reads the same input twice. Even if there is an index on the input (in fact, we are going to assume a perfect index that can filter out irrelevant rows), many rows \((R_2, R_3, R_4)\) would still be read twice (because they are needed for both V1 and V2) and processed independently. Unfortunately, SQL’s relational operators provide no better way of expressing the query because there is no way to produce multiple output rows for each input row, other than by using a Union operator as in this example. When the inputs to the Union have a common source, the binary operator induces redundant I/O. This example shows a case where input data is read redundantly, however in general a Union could induce redundant shuffles as well.

There is a better way to implement the query directly using map-reduce operators. Consider the mapper shown in Figure 3. It reads and processes each input row once, producing up to two output rows per input row. The mapper applies the filters (Line 4 and Line 7) one after the other and outputs the relevant columns. (We operate in the standard multi-set

\(^1\)We show queries as a sequence of statements for the ease of illustration. They could have instead be written as a single nested query; our optimizations still apply in the same manner.
Figure 2: A SQL query, example input-outputs, execution plan showing redundant I/O and an optimized plan produced by RESIN.

Figure 3: A mapper that implements the query Figure 2.

Figure 4: A SQL query with an input-output example.
The implementation of an equi-join requires its inputs to be all the columns of the row are derived from a common set (key column for the two shuffles has the same value, and say two shuffle operators redundantly shuffle a row if count, and partitions the input rows according to the key. We respective join operators (then partitioned on the same column (hr pens despite having the best possible indices because some are redundantly scanned. Note that the redundant scan hap-

 signals rows of the GroupBy Join three for the GroupBy has 9 GroupBy a shuffle is introduced before each requires that its inputs be partitioned by the grouping key. So to reduce the amount of data shuffled. Similarly, a GroupBy hr and performs an aggregation on each partition. It necessarily partitions the table (e.g., with signal values v of source tables. For our example, the two rows corresponding to v5 and v10, are redundantly shuffled before the join because they come from the same input row in signals table. Furthermore, as the left and right aggregates are computed separately, the aggregated results for the same city (city a for our example) are computed and shuffled redundantly.

Figure 5(b) is the optimized plan generated by RESIN. It has only 4 stages, each table is scanned once and no redundant shuffles. On a real dataset, a query with this structure (TPCDS Q90 for example) would speedup by 2×.

RESIN eliminates redundant I/O through two key techniques: sub-query fusion and binary operator elimination. Sub-query fusion merges operators from different parts of the query if they process the same data. More formally, given two sub-queries Q1 and Q2, the fusion rule attempts to construct a triple ⟨Q, ResinMap1, ResinMap2⟩ such that Q1 = ResinMap1(Q) and Q2 = ResinMap2(Q).

For our example query, RESIN first merges the filters and projects applied on each of the input tables. S1 and S2 is merged into a single RESINMAP operator to obtain S12. Similarly, S3 and S4 are merged into a single RESINMAP operator S12. Notice that the filters are combined with a disjunction and projected columns are unioned, so that all of the data required by the query is read in one go.

The fusion process then recursively moves up the tree and the two joins (J1 and J2) are merged together into a single join (J) that computes both the results. An additional RESINMAP is added right after to ensure that only rows required by either J1 or J2 are retained. A salient feature of the fusions rules is that they ensure that the computation of the fused query Q does not shuffle more rows than the individual queries.

---

2 All the light shaded Filter, Project chains in Figure 5 actually represent RESINMAP. We do not show the RESINMAP explicitly, as we did in Figure 2(d), for ease of exposition.
We use a query language based on SparkSQL [5] to present our analysis formally. We define a table as a multi-set of rows that each follow the same schema. A schema $S$ is a set of pairs of column name and data type: $\{(a_1, t_1), (a_2, t_2), \ldots, (a_n, t_n)\}$. A row $r$ that follows schema $S$ is a tuple of form $\{a_1 : v_1, \ldots, a_n : v_n\}$ that assigns a value $v_i$ of type $t_i$ to column $a_i$ of the schema. In this case, we say $r.a_i = v_i$. We will not explicitly refer to the data-types of columns in the rest of this paper because it is not relevant to our analysis.

### 3.1 SQL Operators

This section defines the core SQL operators of our query language. We assume a generic syntax for expressions that can be evaluated over a row to produce a scalar data value. A predicate is simply an expression that evaluates to a Boolean value. Our implementation supports all SparkSQL expressions and predicates.

**Select** $T_2 = \sigma[\phi](T_1)$

A Select operator discards rows of $T_1$ that do not satisfy the filter predicate $\phi$.

**Project** $T_2 = \pi[map(c_i \leftarrow e_i)](T_1)$

A Project is parameterized by a map of $\langle c_i, e_i \rangle$ pairs, where $c_i$ is a column name and $e_i$ are expressions. Project is a row-wise operator. It iterates over all the rows of the input table $T_1$ and for each row, it applies the expressions $e_i$ to compute data values of output columns $c_i$. Note that Project can be used to create aliases of existing columns. For instance, the operator $\pi[e_{\text{new}} \leftarrow e_{\text{old}}]$ renames input column $c_{\text{old}}$ to the output column $c_{\text{new}}$. We sometimes write this operator as $\pi[C \leftarrow E]$ where $C$ is a list of column names and $E$ is a list of expressions and $||C|| = ||E||$.

**GroupBy** $T_2 = \gamma[K, map(c_i \leftarrow agg_i(col_i))](T_1)$

A GroupBy partitions the input table $T_1$ by unique values of columns $K$ and applies aggregations $agg_i$ over each partition. A partition is also referred to as a group. Each aggregation $agg_i$ applies a commutative and associative function (e.g., sum, min, max, etc.) over a single column $col_i$ of $T_1$. Each column of the output table is either the result of an aggregation or a key column. We sometimes write a GroupBy as $\gamma[K, C \leftarrow A(Col)]$ where $C$ and $Col$ are lists of column names, $A$ is a list of aggregations, and $||C|| = ||A|| = ||Col||$. 

---

We have not shown the output of the fused query; its output is the union of the original fused queries (with extra columns).
Join (equi-join) \( T_2 = \Pi[\psi, j](T_{left}, T_{right}) \)

A Join is a binary operator that matches rows from \( T_{left} \) with rows from \( T_{right} \) on a conjunction of equality predicates \( \psi \) of the form \( (a_1 = b_1 \land a_2 = b_2 \ldots \land a_n = b_n) \), where \( a_i \) are columns of \( T_{left} \) and \( b_i \) are columns of \( T_{right} \). The operator requires that the columns names of the two input arguments be distinct. Parameter \( j \) is a join type and can be any of inner (i), leftOuter (lo), rightOuter (ro), leftSemi (ls), or rightSemi (rs) with the standard semantics [5]. For simplicity we only define rules for inner joins in this paper. Our implementation handles other types as well.

Union \( T_2 = \cup(T_{left}, T_{right}) \)

A Union is a binary operator that unions the rows of \( T_{left} \) and \( T_{right} \). It performs a multi-set union, i.e., it does not remove duplicate rows from the output. The two tables need to have the same number of columns and their types must match. The output table \( T_2 \) retains the schema from the left input. We note that different SQL dialects tend to pick different ways of assigning the output schema of a Union. We choose one particular style that is closest to SPARQL.

A query is a sequence of assignments that each produce a new table from existing ones using one or the operators described above. Formally, let \( T_0, T_1, \ldots, T_n \) be a sequence of input tables. A query is a sequence of assignments of the form \( T_i = \text{uop}(T_j) \) (for unary operators \( \text{uop} \)) or \( T_i = \text{bop}(T_j, T_k) \) (for binary operators \( \text{bop} \)) such that \( i > n, j < i, k < i \). We sometimes refer to a table \( T_i \) by the query that computes it.

3.2 Resin operators

Resin introduces two operators, ResinMap and ResinReduce that are used during the optimization process.

ResinMap \( T_2 = \mu[\text{List}(\phi, C \leftrightarrow E)](T_1) \)

A ResinMap is a row-wise unary operator. It is parameterized by a list \( L \) of \( (\phi, C \leftrightarrow E) \). Its semantics is defined by the imperative code shown in Figure 7. For each input row, the operator can produce up to \( |L| \) output rows. The operator iterates over \( L \) (Line 3), and if the predicate \( \phi \) is satisfied (Line 4) then it applies expressions in \( E \) to compute data values of output columns \( C \) (Line 5). In other words, ResinMap applies different chains of Select (\( \sigma[\phi] \)) followed by Project [\( C \leftrightarrow E \)] operators, to produce multiple output rows for each input row. This operator requires that for each map \( C \leftrightarrow E \) in its list, the set of output columns \( C \) be the same (which is also the schema of the output table). The expressions in \( E \) can, however, be different. For example, \( \mu([(\phi_1, a \leftrightarrow e_1), (\phi_2, a \leftrightarrow e_3, b \leftrightarrow e_4)]) \) is a valid operator, whereas the following is not: \( \mu([(\phi_1, a \leftrightarrow e_1), (\phi_2, c \leftrightarrow e_3)]) \).

ResinReduce \( T_2 = \rho[K, \text{List}(\phi, c \leftrightarrow \text{agg}(col))](T_1) \)

A ResinReduce operator first partitions the input into groups on input columns \( K \), and processes each group independently in a streaming manner. The operator is parameterized by a list \( L \) of triples \((\phi, c, \text{agg}(col)) \). Figure 8 describes the per-group computation. It takes a single group \( G \) as input, represented as the partition key \( G.key \) and a multi-set of rows \( G.rows \). It first initializes all the aggregations (Line 3) and then iterates over the rows in \( G \) (Line 4). For each row, it applies the filter \( \phi \) (Line 6) and then updates the corresponding aggregate \( \text{agg} \). Once the entire group is processed, we get a single row containing the keys and the computed aggregates. As notational convenience, we use \( \text{init}(\text{agg}) \) to denote the identity value for an aggregation \( \text{agg} \). For instance, \( \text{init}(\text{sum}) \) would be 0, \( \text{init}(\text{max}) \) would be \(-\infty\) and \( \text{init}(\text{min}) \) would be \(\infty\).

ResinSimpleMap \( T_2 = \lambda(\phi, C \leftrightarrow E)(T_1) \)

ResinSimpleMap is a simplified version of ResinMap that produces at most one output row per input row. It applies a single predicate \( \phi \) and if a row satisfies the predicate, it computes output columns \( C \) by applying expressions \( E \). Its semantics is as in Figure 7 with \( L \) having a single element. It represents the most basic form of a mapper that still subsumes a Select and a Project.

4 Resin optimizations

Resin integrates new rules into an existing query optimizer. It leverages the existing rules to perform certain normalizations

```plaintext
1 method ResinMapOperator(T, ϕ[C]) {
2     foreach (row in T) {
3         if (ϕ(row)) {
4             for (i in 1..|E|) out.C[i] ← E[i](row)
5         }
6         output(out)
7     }
8 }
9 }
10 }
11 method ResinReduceOperator(G, ϕ[C], ρ[K]) {
12     foreach ((ϕ, c, agg(col)) in L) {
13         out.c ← init(agg)
14     }
15     foreach (row in G.rows) {
16         if (ϕ) out.c ← agg(out.c, row.col)
17     }
18     output(G.keys, out)
19 }
20 }
```
that increase the scope of the newly introduced rules. We begin this section by stating rule ordering assumptions and then describe the two core optimizations of RESIN, namely sub-query fusion and binary operator elimination.

### 4.1 Assumptions

RESIN assumes that the following two rules are applied before further optimizations are attempted. These assumptions are not fundamental to the analysis, they are only required to simplify the presentation.

**Column name normalization** The query language allows the reuse of column names within a query. For example, \( T' = \gamma[a, b \leftarrow \text{sum}(b)](T) \) assigns the column name \( b \) to the result of an aggregation in \( T' \), even though \( b \) is already a column in \( T \). We assume that a normalization pre-pass assigns unique names to new columns produced in the query. For example, the above would be rewritten to \( T' = \gamma[a, b+1 \leftarrow \text{sum}(b)](T) \). Such a pre-pass is commonly applied by all query optimizers. In addition to aggregations, a Project operator can also produce new columns. We require that for any projection map \( \lambda[c_i \leftarrow e_i] \), either \( e_i \) is just \( c_i \) or \( c_i \) is a fresh column name. In other words, either a column is just passed through or the output table must use a fresh column name.

**Predicate pushdown** The optimizer pushes Select operators to apply before Project operators. Such a rewriting is always possible, and in fact, standard optimizers have many rules that ensure Select operators apply on the input data as soon as possible. In particular, RESIN assumes that a Select operator is never a parent of a Project.

We also define some standard functions. The function \( \text{cols}(e) \) takes as input an expression \( e \) and returns the set of column names used in the expression. For example, \( \text{cols}(b_1 + b_3 > 0) = \{b_1, b_3\} \). We also define a function \( \text{fresh}() \) that returns a fresh (globally unique) column name each time. Finally, as the output of a Union operator inherits column names from the left argument, we assume the availability of an expression-renaming function \( \alpha\langle\psi(T_{\text{left}}, T_{\text{right}}), e\rangle \) that given an expression \( e \) over columns of \( T_{\text{right}} \), returns an expression over the corresponding columns of \( T_{\text{left}} \). For example, if \( T_{\text{left}} \) has columns \( \{a_1, a_2, a_3\} \) and \( T_{\text{right}} \) has columns \( \{b_1, b_2, b_3\} \), then \( \alpha\langle\psi(T_{\text{left}}, T_{\text{right}}), b_1 + b_3 > 0\rangle \) is \( a_1 + a_3 > 0 \). We drop the first argument of \( \alpha \) when it is clear from the context.

### 4.2 Generalized sub-query fusion

The goal of sub-query fusion is to combine two queries \( Q_1 \) and \( Q_2 \) that operate on the same set of input tables, but may produce different outputs. Fusion produces a common query \( Q \) and two residual RESINSIMPLEMAP operators \( \lambda_1 \) and \( \lambda_2 \) such that \( Q_1 = \lambda_1(Q) \) and \( Q_2 = \lambda_2(Q) \). This ensures that the any redundant computation across \( Q_1 \) and \( Q_2 \) is captured in one single query \( Q \), and only simple map operators (via the residual operators) are needed to get back the original outputs. Furthermore, as would be evident from the way we fuse operators, we ensure that the computation of \( Q \) itself does not require more stages than what is required for computing just one of the sub-queries. Finally, we ensure that \( Q \) does not output any row that is not needed by either \( Q_1 \) or \( Q_2 \). This kind of fusion is, of course, not always possible. The rules below define the conditions under which it is possible and how to combine the queries when possible.

**Identity Invariant.** Given a RESINSIMPLEMAP operator \( \lambda[\phi, \text{map}(c_i \leftarrow e_i)] \), we say that it satisfies the identity invariant if \( e_i \) is simply \( c_i \) for all indices \( i \). This means that the operator carries a subset of the input columns unmodified to the output table. For a set of columns \( C \), we use the shorthand \( \lambda[I(C)] \) to represent such operators, where \( I(C) \) is the identity function on \( C \): \( \{c \leftarrow c \mid c \in C\} \). We will ensure that all residual operators produced as a result of fusion satisfy the identity invariant.

#### 4.2.1 Base rule

The rule for fusing two RESINSIMPLEMAP operators applied on the same table is shown in Figure 9. The RESIN-SIMPLEMAP operators \( \lambda_1 \) and \( \lambda_2 \) apply different filters and projections to the same input table. Fusing these operators is simple, except that we must take care to establish the identity invariant for the residual operators. This is important for recursively fusing more operators up the query tree. The fusion first applies a disjunction of the filters and a union of the projections (\( \lambda_{\text{common}} \)). This ensures that the necessary rows and columns are carried forward. Next, all the residual operators \( \lambda_1 \) and \( \lambda_2 \) need to do is to apply the specific filters for \( Q_1 \) and \( Q_2 \), respectively.

Note that column-name normalization (Section 4.1) guarantees that for any column \( c \), if \( c \in C_1 \) and \( c \in C_2 \) then the column must be passed through from \( T \), i.e., both \( \lambda_1 \) and \( \lambda_2 \) apply the projection \( c \leftarrow c \). This ensures that the projection map of \( \lambda_{\text{common}} \) is well-defined, i.e., it does not include two different mappings for the same output column.
The figure shows how to fuse \( Q_1 \) and \( Q_2 \), given two fusible queries \( Q_1 \) and \( Q_2 \). The shaded circles depict the output of the fusion.

**Figure 11: RESINSIMPLEMAP query fusion.**

### 4.2.2 Recursive fusion of unary operators

Fusion proceeds recursively. For this section, fix the fact that \( \text{FUSE}(Q_1, Q_2) = \langle Q, \lambda_{r1}, \lambda_{r2} \rangle \). As described in Figure 10, our goal is to construct \( \text{FUSE}(op_1(Q_1), op_2(Q_2)) \), where \( op_1 \) and \( op_2 \) are one of RESINSIMPLEMAP (\( \rho \)), GroupBy (\( \gamma \)) or RESINREDUCE (\( \lambda \)). For ease of notation, an operator \( \lambda_r \) always expands to \( \lambda_r[\phi_1, C_r \leftarrow E_1] \).

Recursive fusion of two RESINSIMPLEMAP operators, which subsumes the fusion of Select and Project operators, is shown in Figure 11. Observe that predicates \( \phi_1 \) and \( \phi_2 \) are applied on the output of \( Q \). Further, as the residual operators satisfy the identity invariant, the columns referred in the predicates \( \phi_1 \) and \( \phi_2 \) also come from the result of \( Q \). Therefore, the identity projections in \( \lambda_{r1} \) and \( \lambda_{r2} \) can be dropped and the filters \( \phi_1 \) and \( \phi_2 \) can be conjoined together, and so can \( \phi_1 \) and \( \phi_2 \). Fusion then follows by applying the rule in Figure 9.

The rule for fusing two GroupBy operators is shown in Figure 12. The figures shows two aggregations on the same table, which is the output of \( Q \), except that they first apply their own filter \( s_{\lambda_{r1}} \) and \( \lambda_{r2} \), respectively. (For simplicity, we have shown a single aggregation in each of the GroupBy operators. The case for multiple aggregations extends easily.) The GroupBy operators are on the same key, so we can fuse them into a single RESINREDUCE operator that does the aggregations conditionally as shown in the figure. In addition, the fusion requires two new aggregations \( r_{c1} \) and \( r_{c2} \) that count how often the predicates are satisfied. For the left (respectively, right) group-by to produce any output for a grouping key, at least some rows in the group should satisfy the filter of \( \lambda_{r1} \) (respectively, \( \lambda_{r2} \)). Thus, we need to guard the left (right) output of the fused query with a predicate that ensures that at least one row in the group satisfied the predicate. The new residual operators \( \lambda'_{r1} \) and \( \lambda'_{r2} \) apply the filters \( r_{c1} > 0 \) and \( r_{c2} > 0 \) to only output groups that have at least some rows that satisfy the predicates. The rule extends directly to the fusion of two RESINREDUCE operators as well.

**Figure 12: GroupBy query fusion. Both \( r_{c1} \) and \( r_{c2} \) are fresh column names.**

**Figure 13: Recursive fusion of binary operators.** The figure shows how to extend fusion of two pairs of fusible queries \( Q_1, Q_3 \) and \( Q_2, Q_4 \) (shown in dotted circles), by additional binary operators \( op_1 \) and \( op_2 \). The shaded circles depict the output of such recursive fusion.

**Column Aliasing** Our implementation relaxes the rule’s precondition that grouping keys be exactly the same; even aliasing columns are allowed. That is, columns can be renamed versions of the same column in an earlier table. The same relaxation also applies to the join rule that follows later.
4.2.3 Binary operator fusion

Binary operator fusion is depicted in Figure 13. It defines \texttt{FUSE}(op_1(Q_1, Q_3), op_2(Q_2, Q_4)) using \texttt{FUSE}(Q_1, Q_2) and \texttt{FUSE}(Q_3, Q_4).

Figure 14 shows the rule for fusing two \textit{Join} operators. The rule simply pulls up the residual predicates from before the join to after. Next, it conjoins the residual predicates that are relevant to \((Q_1 \bowtie Q_3)\), namely \(\phi_{r1} \land \phi_{r3}\), to obtain \(\lambda'_{r1}\). Similarly, \(\phi'_{r2} = \phi_{r2} \land \phi_{r4}\). The residual predicates satisfy the identity invariant. However, we still apply the base fusion rule (Figure 9) to push down the common predicate \((\phi_{r1} \land \phi_{r3}) \lor (\phi_{r2} \land \phi_{r4})\). This would eliminate rows that are not needed by \(Q_1 \bowtie Q_3\) or \(Q_2 \bowtie Q_4\), potentially before a shuffle.

Figure 15 shows the rule for fusing two \textit{Union} operators. We only describe a simplified version of the rule where we assume that \(Q_{lt}\) and \(Q_{rt}\) are union-compatible, i.e., they have the same number of columns and their types match. This version is enough to cover the core ideas.

In order to fuse two unions, we need to be able to pull up filters above a union. This poses a challenge as the output has rows from both sides and we want to apply different predicates to the rows from each side. To enable this pull up, we add an additional (fresh) column \textit{side} that tags rows with the side that generated them. This additional column is added by applying \(\lambda'_{lt}\) and \(\lambda'_{rt}\) to \(Q_{lt}\) and \(Q_{rt}\), respectively. The new residual predicates do an additional check to match rows from the appropriate sides. As the union result renames the columns from the right input, \(\lambda'_{rt}\) additionally applies the renaming function \(\alpha\) (defined in Section 4.1).

4.2.4 Operator alignment and exact fusion

The fusion rules described so far only fuse operators of the same type. \texttt{RESIN} also has an auxiliary rule that enables fusion of operators that are preceded by a \texttt{RESIN}_{\text{SIMPLE}} \texttt{MAP} on one side but not on the other. Given \(Q_1\) and \(Q_2\) are fusible, we can fuse the fusion of \(op_1(\lambda(Q_1))\) and \(op_2(Q_2)\), where \(op_1\) and \(op_2\) are fusible according to rules 1-6 above. We do so by adding an empty lambda \(\lambda_{\varphi} = \lambda[\text{true}, \lambda(\ast)]\) as a child of \(op_2\).

We have described the fusion of core SQL operators. Our implementation handles all \texttt{SPARQL} operators, but fusion of other operators is only possible if they have the exact same parameters and apply on the exact same query. We define this \textit{exact} fusion rule as \texttt{FUSE}(op_1(Q_1), op_2(Q_2)) = op_1(Q_1) only if \(op_1 = op_2\) and \(Q_1 = Q_2\). Finally, note that the rules above define the fusion of two sub-queries. Through repeated application of the rules we can fuse any number of sub-queries (say, \(n\)) into a single query with \(n\) residual operators.

4.3 Binary operator elimination

When the two arguments of a binary operators can be fused, \texttt{RESIN} can sometimes eliminate the binary operator altogether. The two elimination rules are defined below.

\textbf{UNION ELIMINATION RULE}

Given a \textit{Union} query \(\upsilon(Q_1, Q_2)\) where \(Q_1\) and \(Q_2\) can be fused such that \(\texttt{FUSE}(Q_1, Q_2) := \langle Q, \lambda_{r1}, \lambda_{r2} \rangle\) then we
The goal of this rule is to substitute a binary join operator with a mapper, which is a row-wise unary operator. This is only possible if the output of the join has already been computed in the fused query. This holds when the join combines the results of a RESINREDUCE query \( p[K, L] \) and is equi-join on \( K \) (modulo aliasing). The rule is shown in Figure 17. Figure 5 shows an example application of this rule.

5 Implementation

We integrated RESIN into a popular state-of-the-art big-data system SPARK [26]. Our optimizations are general and can be applied to other big-data systems [22, 30] as well. We chose SPARK because it is easier to extend [5], has rich code-generation support as well as competitive performance. Moreover, SPARK already performs some low-level I/O optimizations. For instance, it implements exchange reuse [1, 2] that determines if two exchanges are exactly equivalent and skips the duplicate computation. It also implements store-predicate pushdown that pushes down filters and projections to the storage layer [3].

SPARK makes use of the Catalyst query optimizer [5]. Optimization rules in Catalyst are organized into batches. As is standard, logical rules are applied before physical rules. Each physical operator has a pre-defined map-reduce implementation based on a low-level resilient distributed dataset (RDD) API [25]. SPARK uses a whole-stage code generator [23] to efficiently compile all operators in a single stage. We describe key details of our implementation.

**Initiation and termination of RESIN rules** We added all RESIN rules in a batch that executes after the standard optimizations are applied. These rules apply in a single (pre-order) traversal of the query tree. RESIN initiates fusion starting from input table scans. It then moves up the tree fusing operators recursively. The fusion process terminates when none of the fusion rules apply. At this point, RESIN applies the operator elimination rules in cases where the consumers of a fused query share a common parent. After elimination, the resulting query could have zero or more fused sub-queries whose output is consumed more than once, requiring the use of exchange operators, as described next.

**RESIN exchange reuse** The only operator in SPARK whose output can be consumed more than once is an exchange operator. Thus, RESIN introduces an exchange at the reuse points. An exchange is parameterized by a partitioning column. To decide on the partition column, RESIN traverses up along each of the consumers \( C_i \) until it hits an operator that requires partitioning (RESINREDUCE, Join, GroupBy), and identifies a partitioning column \( p_i \) for each consumer. Next, it picks the column \( p_i \) that is required by most consumers (we use random choice to break ties).

**RESIN operators** We added three new logical operators with the structure defined in Section 3. We also add their corresponding physical operators. The physical operator for RESINSIMPLEMAP is just a combination of Select and Project. We added a new physical operator that implements RESINMAP with appropriate whole-stage code-generation support. The physical operator for RESINREDUCE is implemented by carefully extending existing aggregation iterators in SPARK. This allowed us to delegate the handling of different column types and the various associated subtleties in the application of aggregation functions (e.g., null values, type-casting, overflow/underflow, etc.) to routines already present in SPARK. Finally, we added implementation strategies for our opera-
tors. The strategies analyze the logical operators, construct partial aggregates and introduce partitioning operators (for RESINREDUCE), and substitute the logical operators with corresponding physical operators.

6 Evaluation

We evaluated RESIN using the TPCDS benchmark suite, consisting of 104 queries, at scale factors of 1TB and 10TB. The evaluation was done on two different SPARK clusters. We used a cluster with 120 cores and roughly 480GB memory, spread over 10 nodes for evaluating at 1TB scale. For evaluating at 10TB we used a cluster with 480 cores and 1.6TB memory, spread over 34 nodes. The input tables were stored in parquet format. We ran each query 5 times, discarded the first run and took average of the rest. Among the 104 queries, we found that 40 queries have redundant I/O. As mentioned before, the baseline already has basic I/O optimizations. It pushes predicates and projects to the store for all these queries. And it is able to reuse exchanges (usually right after a map stage) even without RESIN optimizations in about half of these queries. In the rest of this section, we focus on these queries alone. We begin by presenting detailed results at 1TB scale and present summary results at 10TB scale in Section 6.4.

6.1 Optimization opportunity

For each query, we identified stages that perform redundant I/O. This was done post-facto by comparing baseline and optimized plans, and determining the baseline stages that were fused together by RESIN. Figure 18 shows the fraction of time spent in these stages relative to the total execution time of the query. The larger the fraction, the greater the optimization opportunity. We find that 40% of the queries spend at least 50% of the time in stages with redundant I/O. We mark these queries as high-impact queries as they have significant potential for improvement. Another 25% spend at least 10% of their time in stages with redundant I/O, and we mark them as medium-impact queries. The remaining (low-impact) queries may have some redundant I/O but eliminating it is unlikely to affect the overall query execution time.

TPCDS queries are over multiple (fact and dimension) input tables. There are 6 large fact tables and several small dimension tables. A deeper inspection of our results revealed that the fraction of time spent in redundant sub-queries is significantly influenced by whether one of these large tables was redundantly processed or not. All queries that have medium or high impact were processing at least one such table multiple times (sometimes even after joining with few other tables).

6.2 Speedup from RESIN optimizations

Figure 19 reports the performance improvements from RESIN on high and medium impact queries. These cover 25% of the entire benchmark suite. As can be seen, RESIN improves the execution time of most of the queries. It achieves an average (geomean) speedup of 1.4× across these queries. RESIN performs particularly well on high-impact queries where it achieves a geomean speedup of 1.6× with some queries speeding up by 6×.

The queries that benefit most (Q9, Q28, Q88, Q75, Q31, Q90) are also ones where RESIN was able to apply binary operator elimination. All the other queries benefit only from generalized sub-query fusion. Some of these (Q65, Q61, Q81, Q1, Q30, Q59) had multiple exchanges after fusion on the reuse exchange column and they see moderate gain. A few queries (Q92, Q32, Q16,Q41) had reuses close to input scans. These are the queries that see the least benefit because the baseline already performs some basic I/O optimizations (exchange reuse and store-predicate pushdown; see Section 5).

In two queries (Q74, Q41) the data overlap between the sub-queries that were fused was very low. However, fusion still helps produce execution plans with fewer stages, and does so while guaranteeing that the number of rows shuffled after fusion is no more than the baseline. We find that, in Q74, simplifying the plan has some second order system effects (see Section 6.3), and fusion improves performance. In Q41,
the reuse is close to the input and hence fusion only eliminates one map stage. As a result, we see a small 3% degradation.

Comparison with BLITZ. We evaluated BLITZ on these queries and found that it only optimizes two of the queries: Q9 and Q28. Both these queries perform a chain of joins at the end. BLITZ was only able to eliminate the first of these joins and therefore was only able to get speedups of $1.6 \times$ and $1.9 \times$, respectively. This limitation has also been acknowledged in prior work [10]. RESIN eliminates multiple joins and achieves a speedup of $2.4 \times$ and $3.3 \times$, respectively, on these queries.

Speedup on low impact queries. Figure 20 reports speedups for low impact queries. We report the execution time of the entire query along the x-axis. As can be seen RESIN optimizations have no significant gains or degradation on any of these queries. To isolate the effects of RESIN optimizations, we plot the speedup for the sub-query that was optimized. RESIN achieves a moderate speedup on several of these sub-queries. RESIN optimizations show a small degradation in a few of these sub-queries (Q2, Q5, Q95). In Q5 the amount of redundant I/O is too small to matter. In Q2, Q95, the baseline already performs an exchange reuse. RESIN fuses one additional operator, but once again the additional I/O is too small to matter.

Overall, RESIN reduces the total time to run all the 104 queries by 12%. Note that RESIN has a negligible impact on query optimization time; the overall compilation time for the entire benchmark increased from 42 to 45 seconds.

6.3 Impact of RESIN optimizations on systems resources

Figure 21 - Figure 24 plot the impact of RESIN optimizations on disk, network, memory and CPU for medium and high impact queries (we see no discernible impact on low impact queries). For disk, we report the cumulative bytes of data accessed from disk. For network, we report the cumulative number of packet transfers performed. Note that data sizes transferred over the network follow the same trend as disk I/O, as most I/O in a big-data setting is over the network. For memory, we plot the cumulative memory footprint. For CPU, we plot the total CPU time spent by all tasks on all machines. This is a measure of the total CPU work done to evaluate a set of queries and is largely independent of cluster size [19]. We infer the following conclusions from these plots.

First RESIN reduces the cumulative CPU, network and disk footprint, consuming 24%, 25% and 19% fewer resources respectively. The savings in-terms of CPU are slightly higher than disk because RESIN not only saves on I/O but also on I/O induced processing (compression, serialization etc) which have a significant compute cost [14].

Second, RESIN achieves these benefits while incurring the same overall memory cost (Figure 23) as the baseline. A few queries (Q64, Q31, Q61) see a slight increase in memory requirement, while a few others (Q4, Q75, Q88) need lesser memory. However, all these queries see significant reduction in execution time. Overall even if fusion increases the amount of data processed by each operator, it does not impact the overall memory footprint of the workload (see Figure 19).

Third, the gap between RESIN and the baseline widens as we move to the right. Queries on the right usually have deeper operator trees and this graph demonstrates that RESIN is able to fuse deep and complex queries.

Finally, the plots indicate that RESIN optimizations
are fairly robust, even the worst performing queries ($Q_{92}, Q_{32}, Q_{41}$) do not show any discernible degradation on any of the system metrics. In $Q_{74}$ RESIN fusion does not reduce the amount of disk I/O, but it still reduces the CPU and network load, and hence sees an execution time benefit.

### 6.4 Impact on larger scale data

We report the impact of RESIN on TPCDS at 10TB scale. Figure 25 shows the speedup’s obtained for the 40 affected queries. We see that RESIN does somewhat better at larger scale. It obtains higher speedup on a few medium and high impact queries ($Q_{64}, Q_{39a}, Q_{39b}, Q_{28}$) while achieving similar speedups for the other queries (except $Q_{59}$). We find that the average (geomean) speedup for high and medium impact queries goes up to $1.5 \times$ (was $1.4 \times$ at 1TB). Once again, the optimizations have no significant improvement or degradation on the low-impact queries. Figure 26 reports the I/O savings. The total disk I/O saved went up to 31% (was 19% at 1TB). Overall, RESIN reduces the execution time of the entire workload (104 queries) by 17%.
7 Related Work

We discuss three broad lines of work related to this paper.

Advances in big-data query optimization Big-data query optimizers borrow and build upon rewrite rules from the database literature. Several big-data-specific optimizations have also been used [8, 9, 15, 16, 28–30]. However, none of these logically fuse multiple operators or eliminate binary operators. The work that is most closely related to RESIN is BLITZ [10], which presented an extension to the query optimizer to find and substitute sub-queries that can be implemented by a streaming operator. BLITZ added new rules that optimize three specific query patterns. Two of these patterns were self-joins and self-unions that followed a GroupBy. The third pattern was a specialized implementation of a min aggregation followed by a Join. The BLITZ rules can perform some of the operator eliminations that RESIN can perform. However, we find that BLITZ patterns cover a very small fraction of queries in standard benchmarks. Only one of the patterns applies to TPCDS queries and that too only on two queries. Furthermore, BLITZ operators do not compose with each other and therefore do not even eliminate redundant shuffles from multi-way self-joins and self-unions. RESIN introduces the ability to fuse multi-input sub-queries and eliminate unnecessary shuffles. This fusion facilitates more join and union elimination.

Multi-query optimization Multi-query optimization (MQO) is a well studied problem in classical database literature [11, 17, 20, 31]. The goal of MQO is to optimize many concurrently submitted queries together, and is typically done by reusing results of common sub-queries. Such optimizations are typically performed in a single scale-up database setting and trade-off latency for throughput. The goal of RESIN is very different. RESIN looks for intra-query redundancy in the big-data setting, and eliminates it while ensuring no additional rows are shuffled. Thus, it simultaneously improves both latency and throughput. The fusion techniques proposed here are also significantly different than MQO. MQO is typically limited to Select-Project-Join (SPJ) queries, whereas RESIN supports composite fusion for all SparkSQL operators. Such support is necessary to eliminate redundancy from deep queries. Our evaluation reveals that optimization of the high and medium impact queries in TPCDS requires fusion of a large number of operators: 21 of 25 queries have 10 to 30 operators. We show that fusion and elimination are not always possible without having new operators and propose RESINMAP and RESINREDUCE operators to enable this. For example, Union elimination is only possible with RESINMAP and GroupBy fusion is only possible with RESINREDUCE. Finally, our binary operator elimination rules are not part of any multi-query or database optimizer.

Code generation techniques for query processing There is a long line of work on compilation techniques to generate efficient single-machine code for a chain of SQL operators [6, 12, 13, 23]. Such compilers target low level inefficiencies such as virtual call overheads and computation of common sub-expressions across operators. This is an active area of research, and includes recent efforts like FLARE [6] that target the compilation of SPARK to single machine systems. Such compilers have limited scope in the big-data setting because they only optimize the code within a single stage [6]; determining what operators constitute a stage is still decided by the query optimizer. SPARK makes use of one such code-generation engine [23] that builds upon HyPer [13]. The physical operators that we add are whole-stage code-gen enabled and benefit directly from such techniques.

Recent literature has seen advance techniques that optimize mixed-mode queries: queries that embed non-SQL functions and expressions into SQL [7, 16, 24]. This line of work is orthogonal to RESIN.

8 Conclusions

The cost of running big-data queries is dominated by I/O. This paper proposes RESIN, a system that helps identify and eliminate redundant I/O. The system proposes extensions to big-data query optimizers that enable first class map-reduce reasoning during query compilation. We show how these can be used to fuse operators processing overlapping data into a single stage of computation, and sometimes eliminate expensive binary operators altogether. We demonstrate that the optimizations are useful for 40% of queries in TPCDS, and bring significant gains (average 1.4×) to a quarter of the benchmark queries.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Wenguang Chen for their valuable feedback and suggestions. We would also like to thank Ajith Shetty, Srinivas T, Shahid K, Lev Novik and Tomas Talius for code and design reviews.
References


