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

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

Partho Sarthi, Kaushik Rajan, Akash Lal
Microsoft Research India

Abhishek Modi, Prakhar Jain, Mo Liu, Ashit Gosalia
Microsoft

Saurabh Kalikar[*]
Intel

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. Data is materialized to disk at the end of each stage and transferred between stages using an all-to-all network *shuffle* (also referred to as an *exchange*). In practice, shuffles that involve very large amount of data require multiple rounds of I/O in order to incrementally aggregate data [15,27]. It is not surprising therefore, that the cost of running a query is dominated by disk and network I/O [15].

Despite the bottleneck on I/O, we find that state-of-the-art query optimizers [4, 5, 18, 21, 22] produce execution plans that read or shuffle the same data redundantly multiple times. In fact, on a standard benchmark like TPCDS, the SPARK query optimizer produces plans where 40% of queries incur redundant I/O (Section 6). A large fraction of these spend at-least half their time in stages with redundant I/O.

A standard big-data query optimizer (Figure 1) performs query optimization using a sequence of tree-rewrite rules. It applies *logical* rules to substitute operator trees with equivalent trees. Then it uses implementation strategies (also called *physical* rules) to transform an optimized operator tree into a tree of physical operators. Each physical operator has a predefined map-reduce implementation. As shown in the figure, a standard optimizer only performs SQL-to-SQL rewrite rules at the logical level and the physical operators just provide data-parallel implementations of SQL operators.

Performing optimization at the SQL level is not optimal for a runtime that can execute arbitrary data-parallel operators. A previous system called BLITZ [10, 19] shows evidence of this opportunity for further optimization. BLITZ [19] uses program synthesis to identify single-input single-output sub-queries that can be implemented by a single imperative map-reduce program. Through program synthesis, it finds map-reduce implementations of queries where a query optimizer produces inefficient execution plans. Subsequent work [10] added some of the newly discovered operators (referred to as *super-operators*) back into the optimizer along with rewrite-rules that target them. They showed that queries that use a super-operator can run up to $2\times$ faster.

A key limitation of BLITZ, however, is that it only optimizes single-input sub-queries, and further it only targets optimiza-

[*]Work was done while the author was at Microsoft

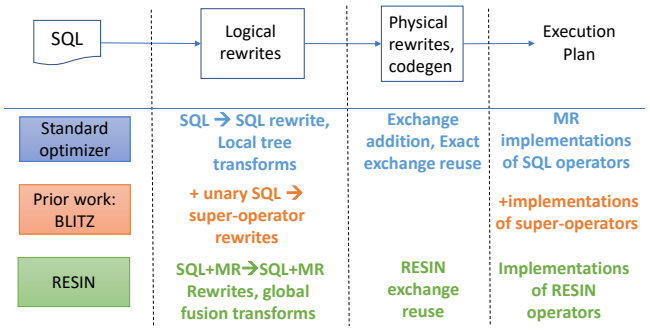

Figure 1: Key components of a big-data query optimizer. RESIN performs map-reduce reasoning starting at the logical level, when prior work only considers it late in the process.

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-unions*. 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 $\langle id, hr_1, signal_1 \rangle$ and $\langle id, hr_2, signal_2 \rangle$ 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_5) 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.

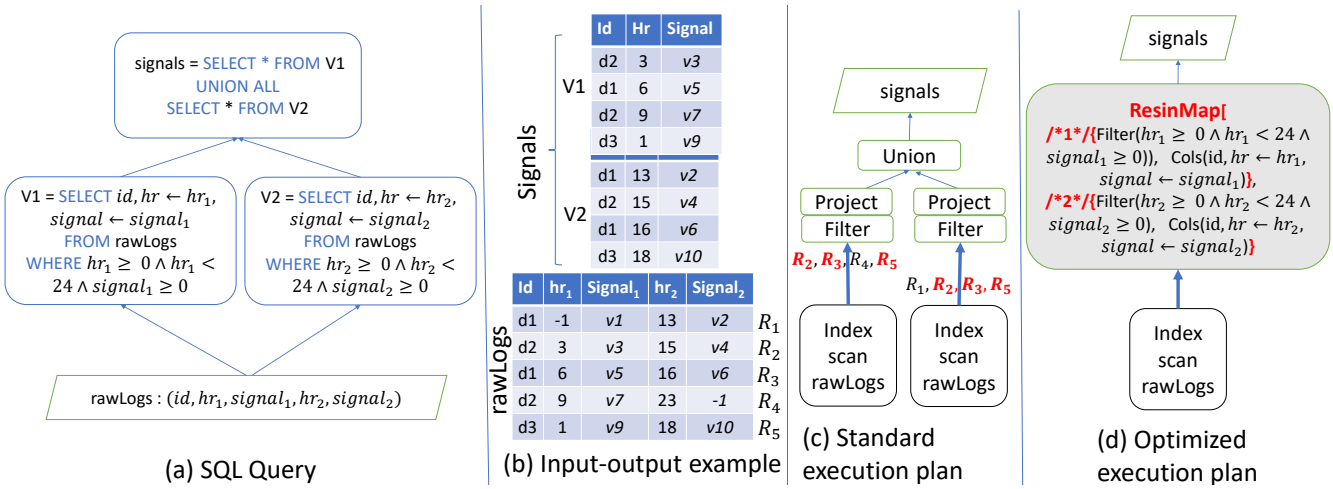

Figure 2: A SQL query, example input-outputs, execution plan showing redundant I/O and an optimized plan produced by RESIN.

```
1  //Mapper m processes a partition rawlogs[m]
2  method exampleResinMap(m){
3    foreach ⟨id,hr1,signal1,hr2,signal2⟩ ∈ rawLogs[m] {
4      if (hr1 ≥ 0∧hr1 < 24∧signal1 ≥ 0) {
5        hr = hr1; signal = signal1; output(id,hr,signal);
6      }
7      if (hr2 ≥ 0∧hr2 < 24∧signal2 ≥ 0) {
8        hr = hr2; signal = signal2; output(id,hr,signal);
9  } } }
```

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

semantics of SQL, so the order of rows in an output table is immaterial.) The mapper is sufficient to implement our example query. The reason current optimizers do not consider this option is that they do not reason at the level of mappers (or reducers) during optimization. They only reason about SQL operators and perform SQL-to-SQL query rewrites.

RESIN extends the optimizer with a generic map operator RESINMAP and a generic reduce operator RESINREDUCE (used in the next example). RESINMAP is a row-wise operator that may produce zero or more output rows for each input row. The generated plan with RESIN for our example query is shown in Figure 2(d). The plan uses a RESINMAP operator. (The code generated for this operator is essentially the one in Figure 3.) RESINMAP consists of multiple entries (two in the example, marked 1 and 2 in the Figure), each with a filter and associated expressions to produce output. The RESINMAP operator is quite powerful. It can implement any single-input single-output sub-query containing arbitrary combinations of *Select*, *Project* and *Union* operators.

Example 2 As a second example, consider the more complex query shown in Figure 4. The query has two inputs, the *signals* table, which comes from the output of the previous example, and another table called *dInfo* that has device-specific

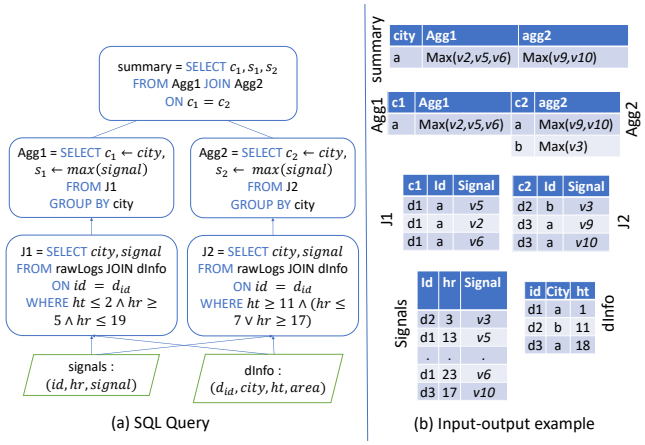

Figure 4: A SQL query with an input-output example.

information. Each row of *dInfo* contains a device identifier d_{id}, the *city* of deployment of the device, and the height *ht* at which the device is installed. The query works as follows. Its result is the *Join* of two intermediate tables $Agg1$ and $Agg2$. The table $Agg1$ contains the maximum day-time reading ($5 \leq hr \leq 19$) per city among all devices deployed at ground level ($ht \leq 2$). This itself requires a join on the *signals* and *dInfo* tables. The table $Agg2$ similarly is the maximum night-time ($hr \geq 17 \vee hr \leq 7$) reading for devices deployed at a height above the ground level ($ht \geq 11$).

A *Join* operator is parameterized by a predicate in the *ON* clause. It takes all combinations of pairs of rows from its input tables, concatenates them and filters according to the *ON* condition. A common usage of *Join* is an *Equi-Join*, where the *ON* clause equates the values of one (or more) columns in the first argument table with one (or more) columns in the second argument table. The query contains three equi-joins, two on the device identifier (for $J1$ and $J2$) and one on the

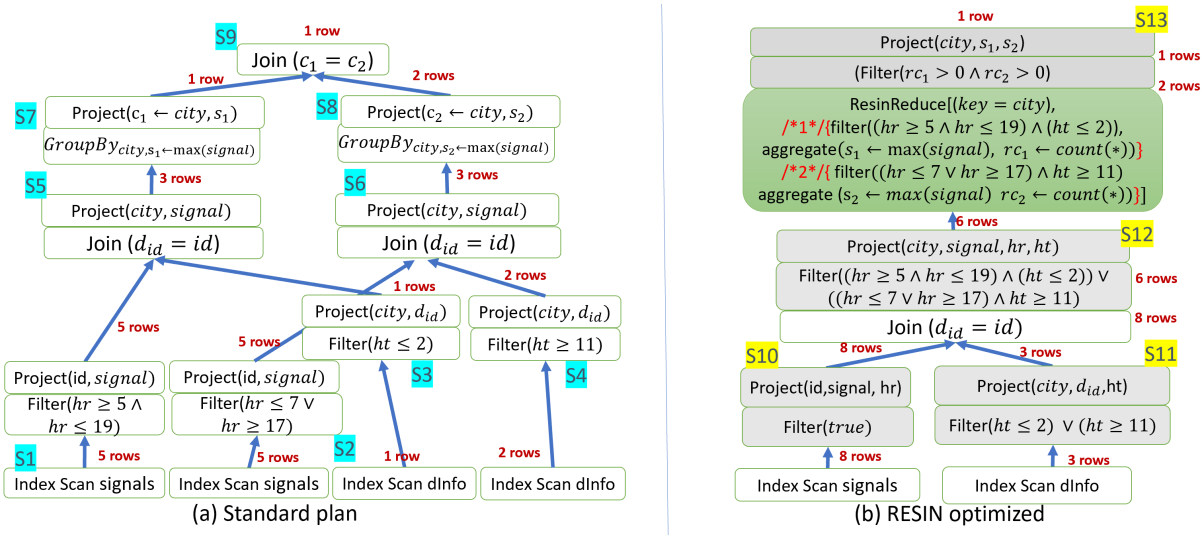

Figure 5: Query execution plans before and after RESIN optimization.

city (for the final output *summary*).

Aggregation happens via the *GroupBy* operator; it partitions its input table on unique values of the grouping key and performs an aggregation on each partition. It necessarily produces only one output row per partition. For instance, the computation of $Agg1$ partitions the table $J1$ on $city$, and for each partition it computes the maximum *signal* value using the aggregation *max*.

Figure 5 shows the execution plan generated by SPARK for this query (solid lines indicate I/O). As before, there is an index scan used each time the query references an input table. The implementation of an equi-join requires its inputs to be partitioned on equated columns, so shuffles are introduced along both arguments for all of the joins. Standard predicate push-down rules in a query optimizer will push the filters on *hr* and *ht* below join and into the respective scans in order to reduce the amount of data shuffled. Similarly, a *GroupBy* requires that its inputs be partitioned by the grouping key. So a shuffle is introduced before each *GroupBy*. In total, the plan has 9 stages; four for scanning input tables ($S1$, $S2$, $S3$, $S4$), three for the *Join* operators ($S5$, $S6$, $S9$), and a further two for the *GroupBy* operators ($S7$, $S8$).

This plan has many sources of redundant I/O. First, some rows of the *signals* table (e.g., with signal values $v5$ and $v10$) are redundantly scanned. Note that the redundant scan happens despite having the best possible indices because some rows can satisfy both the filters on *hr*. The scanned tables are then partitioned on the same column (*id*) and shuffled to the respective join operators ($J1$ and $J2$). A shuffle is a partitioning operator, it takes as input a partitioning key and a partition count, and partitions the input rows according to the key. We say two shuffle operators redundantly shuffle a row if (a) the key column for the two shuffles has the same value, and (b) all the columns of the row are derived from a common set 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 $2\times$.

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 $\langle Q, ResinMap_1, ResinMap_2 \rangle$ such that $Q_1 = ResinMap_1(Q)$ and $Q_2 = ResinMap_2(Q)$.

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 $S11$[2]. 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.

```
1  // ps ∈ partition(J, [city,signal,ht,hr]);
2  max1 = max2 = ∞;
3  rc1 = rc2 = 0;
4  foreach (⟨city,signal,ht⟩ in ps.group) {
5    if (hr ≤ 19 ∧ hr ≥ 5 ∧ ht ≤ 2 {
6      max1=max(max1,signal); rc1 = rc1 + 1;
7    }
8    if ((hr ≥ 17 ∨ hr ≤ 7) ∧ ht ≥ 11) {
9      max2=max(max2,signal); rc2 = rc2 + 1;
10   }
11 }
12 output (city, max1, max2, rc1,rc2);
```

Figure 6: A reducer for the query plan in Figure 5.

Note once again that the output of the individual joins $J1$ and $J2$ can be separated out by applying appropriate filters on J. The reader can verify that $J1 = Select(hr \leq 19 \wedge hr \geq 5 \wedge ht \leq 2)$ *from* J and $J2 = Select((hr \geq 17 \vee hr \leq 7) \wedge ht \geq 11)$ *from* J.

Finally, the two aggregations are fused together. Note that the aggregates need to be applied on different filtered subsets of J. It turns out that fusion of aggregation operations cannot be done with standard SQL operators, neither with RESINMAP alone. RESIN *introduces a generic reduce operator* RESINREDUCE *that makes the optimization process much more expressive by directly considering map-reduce plans.* As seen in Figure 5, the two *GroupBy* are fused into a RESINREDUCE operator. The RESINREDUCE operator is parameterized by a (partition) key (in this case, *city*) and a list with two entries. Each entry contains the filter that determines a subset of rows to be aggregated as well as the aggregation function. In addition, each entry has a $count(*)$ aggregation for reasons described below.

Figure 6 shows the code that implements the reducer, applied to each partition of J (partitioned on *city*) independently. The reducer maintains variables max_1, max_2, rc_1, rc_2 for computing four aggregations. It then iterates over the partition, and for each row, it updates the aggregation variable if the row satisfies the corresponding predicate. The reducer outputs one row per partition, with five columns: the grouping key *city* and the four aggregated values.

The variables rc_1 and rc_2 are used to check if an aggregation was even applied for a partition. This is necessary to obtain back the output of the original *GroupBy*s. For our example, the output of the reducer[3] will contain 2 rows corresponding to cities a and b, however for $city = b$, $rc_1 = 0$, indicating that $Agg1$ has no output for $city = b$.

Once the aggregations are fused, RESIN performs *binary operator elimination* to get rid of the final *Join* ($S7$) altogether. RESIN figures out that the join was doing nothing more than putting together the aggregates from the two sub-queries, which is already done in the output of the RESINREDUCE operator. RESIN replaces the *Join* with a simple filter that ensures that a row is output only if both aggregates produce an output, as is dictated by the semantics of a join.

In summary, RESIN introduces a class of optimizations that target map-reduce operators to eliminate redundant I/O.

[3] We have not shown the output of the fused query; its output is the union of the original fused queries (with extra columns).

3 Preliminaries

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: $\{\langle a_1,t_1\rangle\langle a_2,t_2\rangle\cdots,\langle a_n,t_n\rangle\}$. A row r that follows schema S is a tuple of form $\{a_1 : v_1,\cdots,a_n : v_n\}$ 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 ϕ.

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 $\pi[c_{new} \leftarrow c_{old}]$ renames input column c_{old} to the output column c_{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 $\|C\| = \|A\| = \|Col\|$.

```
1  method ResinMapOperator(T, μ[L]) {
2    foreach(row in T) {
3      foreach(⟨ϕ,C ← E⟩ in L) {
4        if(ϕ(row)){
5          for(i in 1..|E|) out.C[i] ← E[i](row)
6          output(out)
7        }
8      }
9    }
10 }
```

Figure 7: RESINMAP Operator

```
1  method ResinReduceOperator(G, ρ[K,L]) {
2    foreach (⟨ϕ_i,c_i,agg_i(col_i)⟩ in L)
3      out.c_i ← init(agg_i)
4    foreach(row in G.rows) {
5      foreach (⟨ϕ_i,c_i,agg_i(col_i)⟩ in L)
6        if (ϕ_i)  out.c_i = agg(out.c_i,row.col_i)
7    }
8    output(G.keys,out)
9  }
```

Figure 8: RESINREDUCE Operator

Join (equi-join) $T_2 = \bowtie[\psi, jt](T_{left}, T_{right})$

A *Join* is a binary operator that matches rows from T_{left} with rows from T_{right} on a conjunction of equality predicates ψ of the form $(a_1 = b_1 \wedge a_2 = b_2 \ldots \wedge a_n = b_n)$, 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 *jt* 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 $T_2 = \uplus(T_{left}, T_{right})$

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 SPARKSQL.

A *query* is a sequence of assignments that each produce a new table from existing ones using one the operators described above. Formally, let $T_0, T_1, \cdots, T_n$ be a sequence of input tables. A query is a sequence of assignments of the form $T_i = \texttt{uop}(T_j)$ (for unary operators `uop`) or $T_i = \texttt{bop}(T_j, T_k)$ (for binary operators `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 $T_2 = \mu[List(\phi, C \leftarrow E)](T_1)$

A RESINMAP is a row-wise unary operator. It is parameterized by a list L of pairs $\langle \phi, C \leftarrow E \rangle$. 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 ϕ 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*$[C \leftarrow E]$ operators, to produce multiple output rows for each input row. This operator requires that for each map $C \leftarrow E$ 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, $\mu[(\phi_1, a \leftarrow e_1, b \leftarrow e_2), (\phi_2, a \leftarrow e_3, b \leftarrow e_4)]$ is a valid operator, whereas the following is not: $\mu[(\phi_1, a \leftarrow e_1), (\phi_2, c \leftarrow e_3)]$.

RESINREDUCE $T_2 = \rho[K, List(\phi, c \leftarrow agg(col))](T_1)$

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 $\langle \phi, c, agg(col)) \rangle$. 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 ϕ_i (Line 6) and then updates the corresponding aggregate agg_i (Line 6). Once the entire group is processed, we get a single row containing the keys and the computed aggregates. As notational convenience, we use $init(agg)$ to denote the identity value for an aggregation agg. For instance, $init(sum)$ would be 0, $init(max)$ would be $-\infty$ and $init(min)$ would be ∞.

RESINSIMPLEMAP $T_2 = \lambda[\phi, C \leftarrow E](T_1)$

RESINSIMPLEMAP is a simplified version of RESINMAP that produces at-most one output row per input row. It applies a single predicate ϕ 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

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, $T' = \gamma[a, b \leftarrow sum(b)](T)$ 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 $T' = \gamma[a, b\#1 \leftarrow sum(b)](T)$. 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 $map(c_i \leftarrow e_i)$, 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 $cols(e)$ takes as input an expression e and returns the set of column names used in the expression. For example, $cols(b_1 + b_3 > 0)$ is $\{b_1, b_3\}$. We also define a function $fresh()$ 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 $\alpha(\uplus(T_{left}, T_{right}), e)$ that given an expression e over columns of T_{right}, returns an expression over the corresponding columns of T_{left}. For example, if T_{left} has columns (a_1, a_2, a_3) and T_{right} has columns (b_1, b_2, b_3), then $\alpha(\uplus(T_{left}, T_{right}), b_1 + b_3 > 0)$ is $a_1 + a_3 > 0$. We drop the first argument of α when it it 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 λ_{r1} and λ_{r2} such that $Q_1 = \lambda_{r1}(Q)$ and $Q_2 = \lambda_{r2}(Q)$. 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.

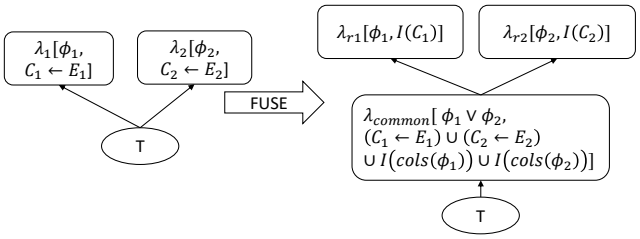

Figure 9: Basic query fusion.

Identity Invariant. Given a RESINSIMPLEMAP operator $\lambda[\phi, map(c_i \leftarrow e_i)]$, 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 $\lambda[\phi, I(C)]$ 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 RESINSIMPLEMAP operators λ_1 and λ_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 (λ_{common}). This ensures that the necessary rows and columns are carried forward. Next, all the residual operators λ_1 and λ_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 λ_1 and λ_2 apply the projection $c \leftarrow c$. This ensures that the projection map of λ_{common} is well-defined, i.e., it does not include two different mappings for the same output column.

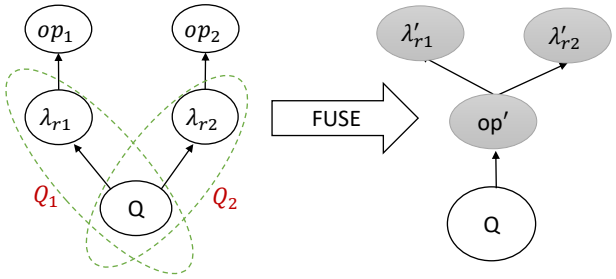

Figure 10: Recursive fusion of unary operators. Given two fusible queries Q_1 and Q_2, shown in dotted circles, the figure shows how to fuse $op_1(Q_1)$ and $op_2(Q_2)$. The shaded circles depict the output of the fusion.

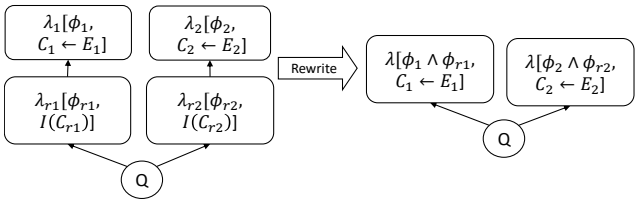

Figure 11: RESINSIMPLEMAP query fusion.

4.2.2 Recursive fusion of unary operators

Fusion proceeds recursively. For this section, fix the fact that $FUSE(Q_1, Q_2) \coloneqq \langle Q, \lambda_{r1}, \lambda_{r2} \rangle$. As described in Figure 10, our goal is to construct $FUSE(op_1(Q_1), op_2(Q_2))$, where op_1 and op_2 are one of RESINSIMPLEMAP (λ), *GroupBy* (γ) or RESINREDUCE (ρ). For ease of notation, an operator λ_x always expands to $\lambda[\phi_x, C_x \leftarrow E_x]$.

Recursive fusion of two RESINSIMPLEMAP operators, which subsumes the fusion of *Select* and *Project* operators, is shown in Figure 11. Observe that predicates ϕ_{r1} and ϕ_{r2} are applied on the output of Q. Further, as the residual operators satisfy the identity invariant, the columns referred in the predicates ϕ_1 and ϕ_2 also come from the result of Q. Therefore, the identity projections in λ_{r1} and λ_{r2} can be dropped and the filters ϕ_1 and ϕ_{r1} can be conjoined together, and so can ϕ_2 and ϕ_{r2}. 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 sλ_{r1} and λ_{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

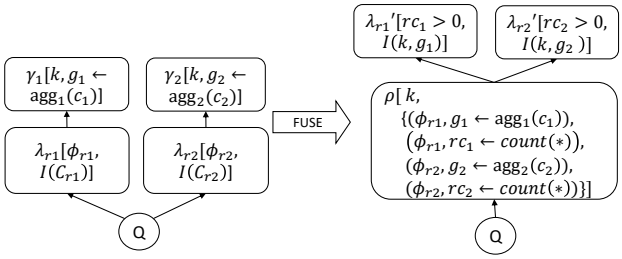

Figure 12: GroupBy query fusion. Both rc_1 and rc_2 are fresh column names.

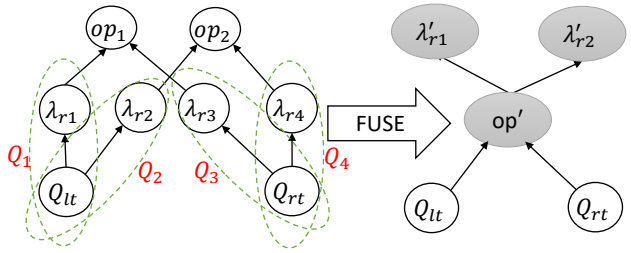

Figure 13: Recursive fusion of binary operators. The figure shows how to extend fusion of two pairs of fusible queries Q_1, Q_3 and Q_2, Q_4 (shown in dotted circles), by additional binary operators op_1 and op_2. The shaded circles depict the output of such recursive fusion.

fusion requires two new aggregations rc_1 and rc_2 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 λ_{r1} (respectively, λ_{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 λ'_{r1} and λ'_{r2} apply the filters $rc_1 > 0$ and $rc_2 > 0$ 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.

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.

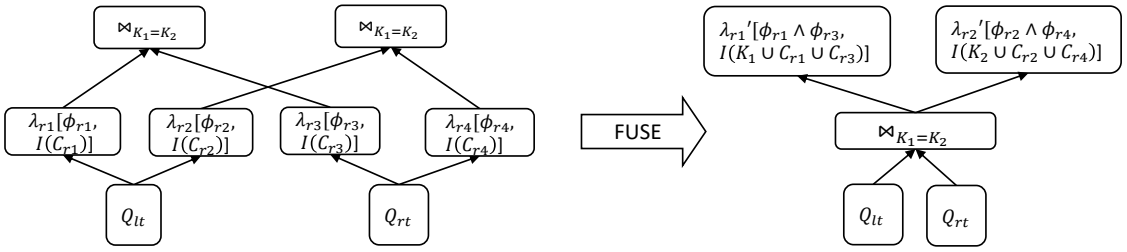

Figure 14: Join query fusion

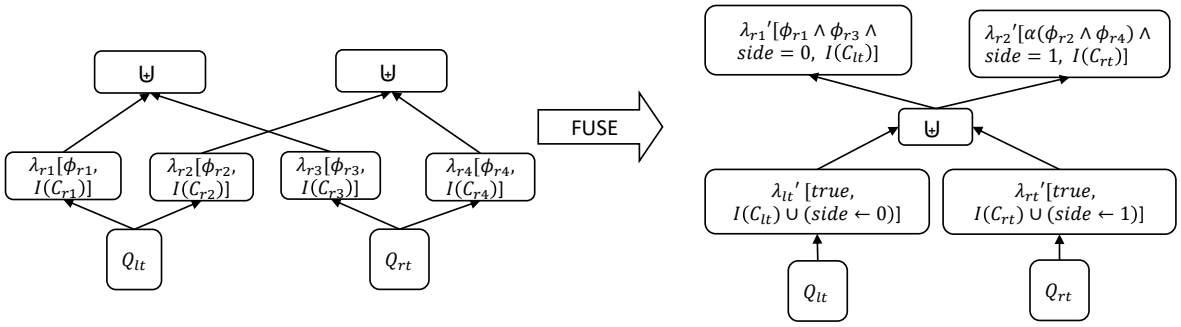

Figure 15: Union query fusion. The column *side* is a fresh name.

4.2.3 Binary operator fusion

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

Figure 14 shows the rule for fusing two *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 ϕ_{r1} and ϕ_{r3}, to obtain λ'_{r1}. Similarly, $\phi'_{r2} = \phi_{r2} \wedge \phi_{r4}$. The residual predicates satisfy the identity invariant. However, we still apply the base fusion rule (Figure 9) to push down the common predicate $(\phi_{r1} \wedge \phi_{r3}) \vee (\phi_{r2} \wedge \phi_{r4})$. This would eliminate rows that are not needed by either $Q_1 \bowtie Q_3$ or $Q_2 \bowtie Q_4$, potentially before a shuffle.

Figure 15 shows the rule for fusing two *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 *side* that tags rows with the side that generated them. This additional column is added by applying λ'_{lt} and λ'_{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, λ'_{r2} additionally applies the renaming function α (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. RESIN also has an auxiliary rule that enables fusion of operators that are preceded by a RESINSIMPLEMAP on one side but not on the other. Given Q_1 and Q_2 are fusible, we enable 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 $\lambda_e = \lambda[true, I(*)]$ as a child of op_2.

We have described the fusion of core SQL operators. Our implementation handles all SPARKSQL 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 *exact* fusion rule as $FUSE(op_1(Q_1),op_2(Q_2)) = op_1(Q_1)$ only if $op1 = op2$ 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, RESIN can sometimes eliminate the binary operator altogether. The two elimination rules are defined below.

UNION ELIMINATION RULE

Given a *Union* query $\uplus(Q_1,Q_2)$ where Q_1 and Q_2 can be fused such that $FUSE(Q_1,Q_2) := \langle Q,\lambda_{r1},\lambda_{r2}\rangle$ then we

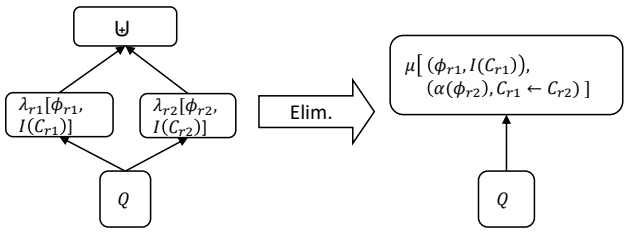

Figure 16: Union elimination rule.

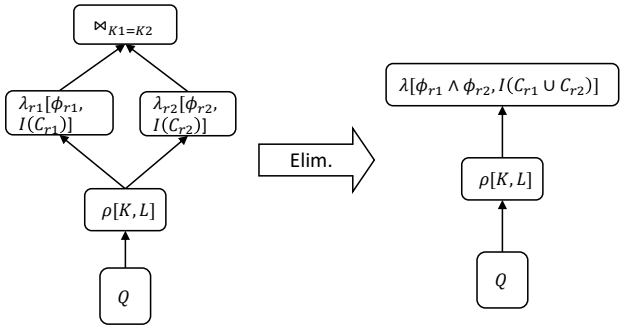

Figure 17: Join elimination rule. The rule requires that each of K_1 and K_2 alias with K.

can eliminate the *Union* altogether using the rule shown in Figure 16. The rule follows directly from the definition of ResinMap. Recall that a ResinMap operator $\mu[List(\phi, C \leftarrow E)]$ can produce multiple outputs per input row. This sufficient to implement the *Union* operator of the form above. The resulting ResinMap operator has one entry for each input that applies the filter ϕ_{ri}. The right expressions are renamed using α and assigned to the column names from the left (C_1) to conform to the semantics of a *Union*. Figure 2 is an example application of this rule.

Resin rules show that any single-input query consisting of *Select*, *Project* and *Union* operators can be implemented by a single ResinMap operator.

Join elimination rule 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 $\rho[K, L]$ 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-gen 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-

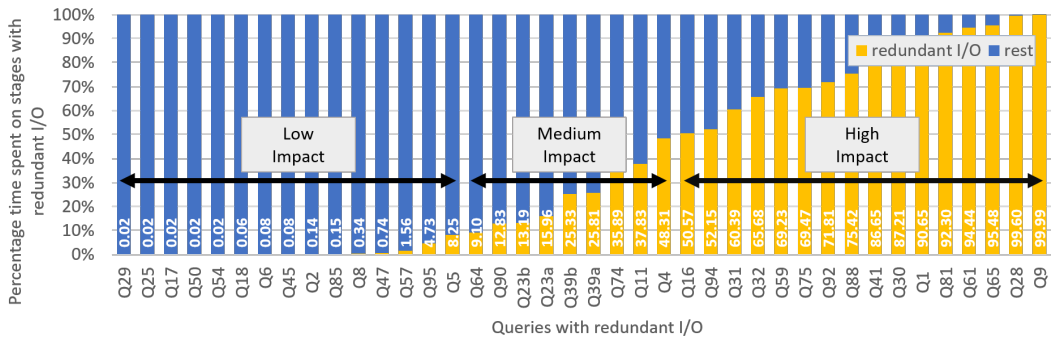

Figure 18: Fraction of time spent in stages with redundant I/O.

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 $Q41$,

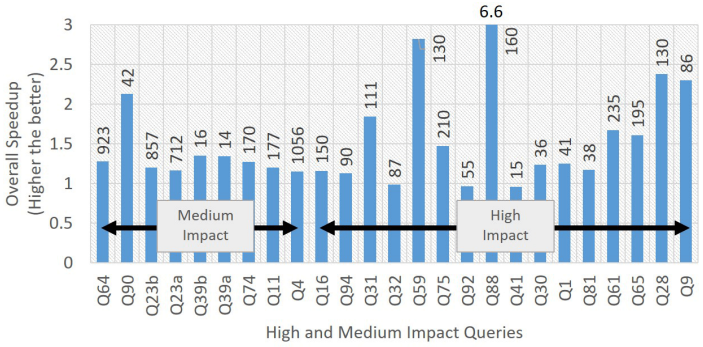

Figure 19: Overall execution time speedup for high and medium impact queries. Each bar is labeled with the execution time of the baseline query (in seconds).

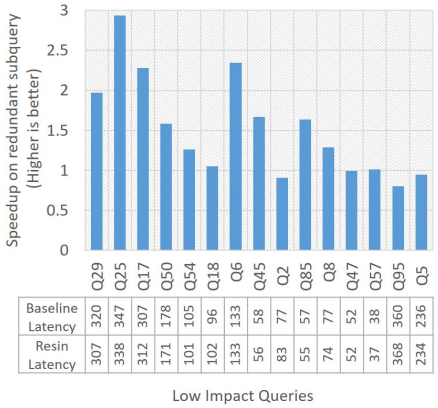

Figure 20: For low impact queries we plot the speedup in sub-query with redundant I/O alone. Along the x-axis we report the execution time of the entire query (in seconds) with and without RESIN.

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: *Q*9 and *Q*28. 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× and 1.9×, respectively. This limitation has also been acknowledged in prior work [10]. RESIN eliminates multiple joins and achieves a speedup of 2.4× and 3.3×, 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 (*Q*2, *Q*5, *Q*95). In *Q*5 the amount of redundant I/O is too small to matter. In *Q*2, *Q*95, 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 (*Q*64, *Q*31, *Q*61) see a slight increase in memory requirement, while a few others (*Q*4, *Q*75, *Q*88) 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

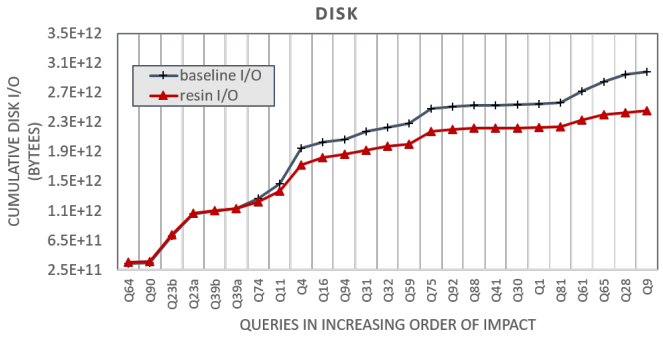

Figure 21: Cumulative disk I/O

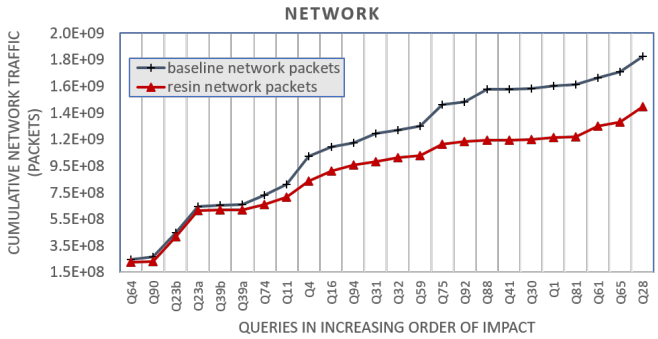

Figure 22: Cumulative network packets

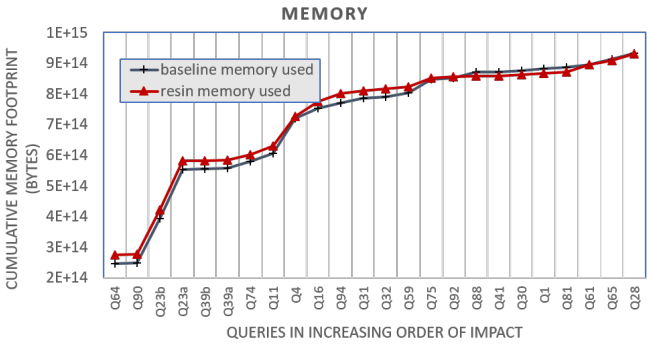

Figure 23: Cumulative memory footprint

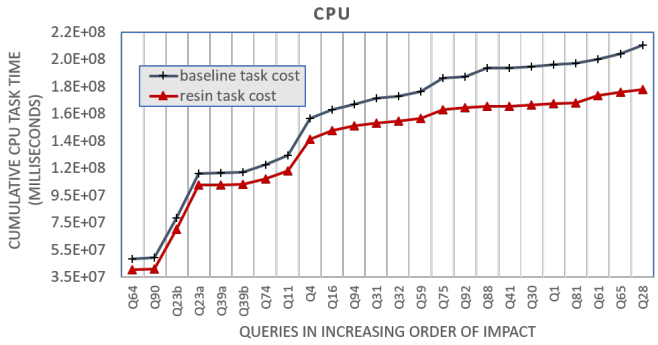

Figure 24: Cumulative CPU time of tasks

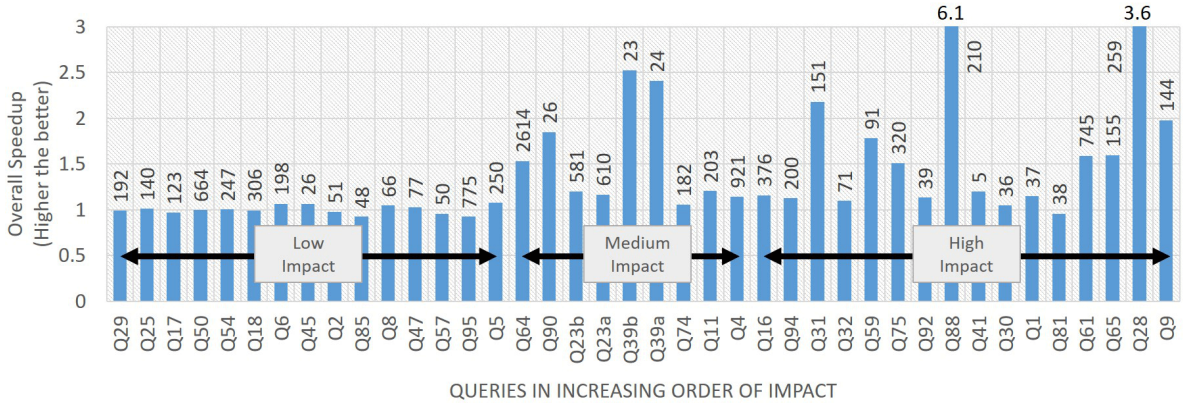

Figure 25: Speedup for 10TB TPCDS. Each bar is labeled with the execution time of the baseline query (in seconds).

are fairly robust, even the worst performing queries ($Q92$,$Q32$,$Q41$) do not show any discernible degradation on any of the system metrics. In $Q74$ 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 ($Q64, Q39a, Q39b, Q28$) while achieving similar speedups for the other queries (except $Q59$). We find that the average (geomean) speedup for high and medium impact queries goes up to 1.5× (was 1.4× 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%.

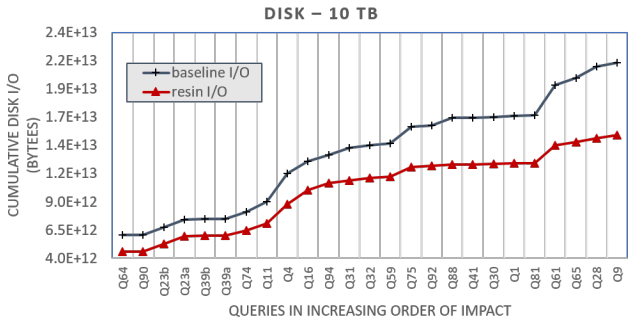

Figure 26: Cumulative disk I/O for 10TB TPCDS.

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* on the same input table. 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 composible 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

[1] Reuse Query Fragments. https://issues.apache.org/jira/browse/SPARK-13756, 2016.

[2] Reuse the exchanges in a query. https://issues.apache.org/jira/browse/SPARK-13523, 2016.

[3] Parquet Predicate Pushdown improvement. https://issues.apache.org/jira/browse/SPARK-25419, 2018.

[4] Amazon red-shift. https://docs.aws.amazon.com/redshift/index.html, 2020.

[5] 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 International Conference on Management of Data, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[6] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In OSDI, pages 799–815, 2018.

[7] X. Fan, Z. Guo, H. Jin, X. Liao, J. Zhang, H. Zhou, S. McDirmid, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. IEEE Transactions on Parallel and Distributed Systems, 26(6):1718–1731, June 2015.

[8] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In OSDI, pages 121–133, 2012.

[9] Yin Huai, Ashutosh Chauhan, Alan Gates, Gunther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major technical advancements in apache hive. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pages 1235–1246, New York, NY, USA, 2014. ACM.

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

[11] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Mqjoin: Efficient shared execution of main-memory joins. Proc. VLDB Endow., 9(6):480–491, January 2016.

[12] Derek Gordon Murray, Michael Isard, and Yuan Yu. Steno: Automatic optimization of declarative queries. In PLDI, pages 121–131, 2011.

[13] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. PVLDB, 4(9), 2011.

[14] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 293–307, Oakland, CA, May 2015. USENIX Association.

[15] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. Hyper dimension shuffle: Efficient data repartition at petabyte scale in. PVLDB, 12(10):1113–1125, 2019.

[16] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In SOSP, pages 153–167, 2015.

[17] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, page 249–260, New York, NY, USA, 2000. Association for Computing Machinery.

[18] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David G Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Allen Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed Aly, Divy Agrawal, Ashish Gupta, and Shivakumar Venkataraman. F1 query: Declarative querying at scale. pages 1835–1848, 2018.

[19] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. Optimizing big-data queries using program synthesis. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 631–646, New York, NY, USA, 2017. ACM.

[20] Timos K. Sellis. Multiple-query optimization. ACM Trans. Database Syst., 13(1):23–52, March 1988.

[21] Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David DeWitt, and César Galindo-Legaria. Query optimization in microsoft sql server pdw. In Proceedings of the 2012 ACM SIGMOD International Conference on

Management of Data, SIGMOD '12, page 767–776, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. Proc. VLDB Endow., 2(2):1626–1629, August 2009.

[23] Reynold Xin and Josh Rosen. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. https://tinyurl.com/mzw7hew, 2015.

[24] Guoqing Harry Xu, Margus Veanes, Michael Barnett, Madan Musuvathi, Todd Mytkowicz, Ben Zorn, Huan He, and Haibo Lin. Niijima: Sound and automated computation consolidation for efficient multilingual data-parallel pipelines. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, pages 306–321, New York, NY, USA, 2019. ACM.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[27] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018. ACM.

[28] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In Proceedings of the Thirteenth EuroSys Conference, EuroSys '18, pages 43:1–43:15, New York, NY, USA, 2018. ACM.

[29] J. Zhou, P. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In ICDE, pages 1060–1071, 2010.

[30] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: Parallel databases meet mapreduce. The VLDB Journal, 21(5):611–636, October 2012.

[31] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, page 533–544, New York, NY, USA, 2007. Association for Computing Machinery.