Incorporating Super-Operators in Big-Data Query Optimizers

Jyoti Leeka
Microsoft
Jyoti.Leeka@microsoft.com

Kaushik Rajan
Microsoft
krajan@microsoft.com

ABSTRACT

The cost of big-data analytics is dominated by shuffle operations that induce multiple disk reads, writes and network transfers. This paper proposes a new class of optimization rules that are specifically aimed at eliminating shuffles where possible. The rules substitute multiple shuffle inducing operators (*Join*, *UnionAll*, *Spool*, *GroupBy*) with a single streaming operator which implements an entire sub-query. We call such operators *super-operators*.

A key challenge with adding new rules that substitute sub-queries with super-operators is that there are many variants of the same sub-query that can be implemented via minor modifications to the same super-operator. Adding each as a separate rule leads to a search space explosion. We propose several extensions to the query optimizer to address this challenge. We propose a new abstract representation for operator trees that captures all possible sub-queries that a super-operator implements. We propose a new rule matching algorithm that can efficiently search for abstract operator trees. Finally we extend the physical operator interface to introduce new parametric super-operators.

We implement our changes in SCOPE, a state-of-the-art production big-data optimizer used extensively at Microsoft. We demonstrate that the proposed optimizations provide significant reduction in both resource cost (average 1.7×) and latency (average 1.5×) on several production queries, and do so without increasing optimization time.

PVLDB Reference Format:
Jyoti Leeka and Kaushik Rajan. Incorporating Super-Operators in Big-Data Query Optimizers. *PVLDB*, 13(3): 348-360, 2019.
DOI: https://doi.org/10.14778/3368289.3368299

1. INTRODUCTION

Today, many companies process several peta-bytes of data every single day using SQL like languages. To meet their data-analysis needs they have either built their own big-data systems or rely on cloud providers to scale compute [26, 29, 32, 15] and storage [5, 10, 22] to data-centers with 1000s of machines. Processing such massive amounts of data on large clusters presents new challenges and opportunities for query optimizers. Like any standard database optimizer, the big-data query optimizer relies on rewrite rules, to search over different equivalent queries and operator implementations, to produce an efficient execution plan. In the big-data setting the optimizer produces a distributed plan by introducing stage breaks in the physical plan. Each stage runs in a data-parallel manner on many machines and data is shuffled between stages over the network. Such shuffles typically dominate the cost of big-data analytics [30, 21], as each shuffle involves multiple disk writes, disk reads and network transfers across many machines.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368299

Optimization opportunity

The goal of our project, BLITZ, is to ask if the current set of rules and operators produce plans with as few shuffles as possible or do they miss opportunities. In previous work we show evidence that state-of-the-art optimizers sometimes produce plans with more shuffles than necessary. We employed program synthesis [24] to look for such queries, and found many instances where sub-queries that were evaluated across several stages could in fact be evaluated by a single data-parallel operator. We found this to be true for all popular big-data query optimizers [26, 32, 4].

We show the optimization opportunity with an example.

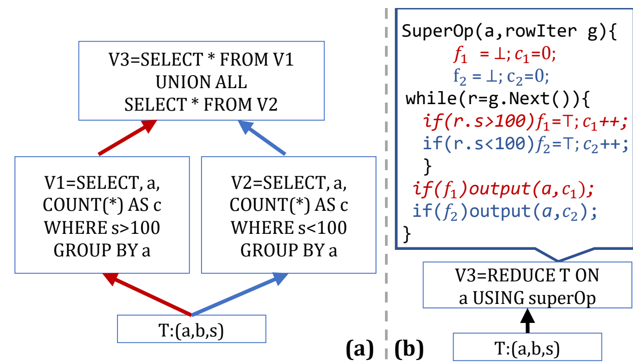

Figure 1: (a) Query with multiple expensive operators (b) Streaming super-operator implementation.

Example 1 *Consider the production query in Figure 1. The query processes cloud store logs gathering summary statistics per account. For each account* $\boldsymbol{a}$ *it captures two statistics, the number of store operations accessing less than*

*100KB of data ($s < 100$) and the number of operations accessing more than 100KB. As shown, the query has a diamond dependence structure, it unions the results of two different views derived from the same table. The views perform grouping aggregations on subsets of the same data, filtered by different predicates. A key property of this query is that it is groupwise [6] in nature, that is every row in the output is only dependent on one partition of the input (in this case on column **a**). We discuss the distributed query plan in detail during evaluation, but in summary it employs 9 stages to compute the output, shuffling 15 terabytes of data.*

*This query was optimized to the following map-reduce form via program synthesis. The mapper partitions the data on column **a**, and the reducer computes multiple filtered aggregates in a single pass over the data. It streams the elements of a single partition through a sequence of guarded commands, exploiting the commutative nature of aggregations to simultaneously compute all the filtered aggregations. The optimized query plan has half the stages, runs* $1.8\times$ *faster, uses half the resources, and shuffles 7TB less data.*

The challenge

This previous work relied on program synthesis, an advanced program reasoning technique that is hard to scale and too slow to perform as part of query compilation. It takes in the order of minutes to produce a result while query compilation takes only seconds. Even to produce output in minutes, several synthesis heuristics had to be used that trade-off correctness guarantees for speed. Overall, while good at discovering such optimization opportunities, our synthesis engine is not guaranteed to produce correct optimizations. Further, as synthesis was itself performed outside the purview of a query optimizer it did not make use of the known equivalences that a query optimizer exploits. As a result there were many instances where synthesis missed optimization opportunities. Despite these limitations our evaluation of synthesized operators across 1000s of queries revealed that when correct, the optimization did provide significant benefits. Once manually inspected to be correct and incorporated into the query as a user defined operator the rewrite resulted in a $1.1\times - 3\times$ improvement in query latency as well as a $10\% - 70\%$ reduction in query cost.

In this paper we describe how we extended a production big-data query optimizer, with new rules and physical operators, so that the same benefits can be achieved with strong correctness guarantees, and without the other drawbacks of program synthesis. As the big-data system in consideration, SCOPE [32], is extensively used across the entire company (millions of queries every week, processing exabytes of data on several clusters with 40000 nodes or more), it is critical to ensure that such optimizations pass the rigorous correctness and optimization time constraints imposed in the production setting. Even though the synthesis based approach rarely produced an incorrect solution ($< 2\%$ of instances), putting it in a production system was not an option.

The SCOPE optimizer is a cascades [11] based optimizer that extends the SQL server optimizer to the distributed setting. It has all the standard equivalence rules and operator implementations from database literature with several enhancements to specialize it to the big-data setting [32, 31, 23, 14, 21]. As with any cost based optimizer, rules are used to generate alternative operator trees and the one with the least cost is chosen for execution.

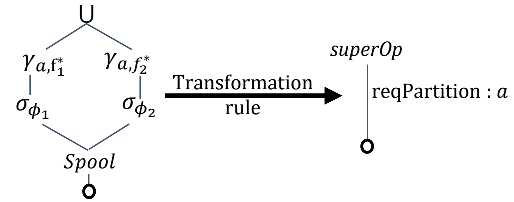

Figure 2: Example rule

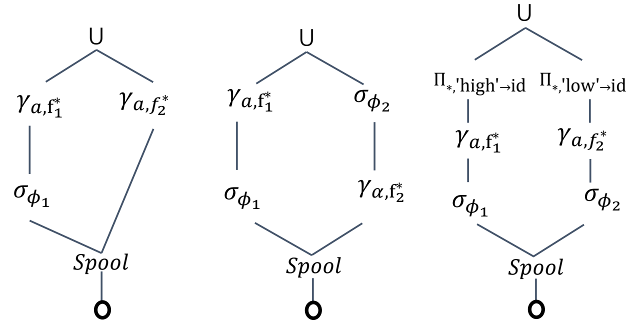

Figure 3: Some variants of the pattern that can be implemented by the same super-operator

The rewrite in Example 1 could be implemented as shown in Figure 2 by introducing a new operator, and a corresponding rule that substitutes the operator tree corresponding to the example sub-query with the new operator. The substituted operator requires data to be partitioned on column a, and processes each partition in a streaming manner. Note unlike a standard operator that implements a single SQL operator, the above operator implements an entire sub-query. We therefore refer to it as a *super-operator*.

A key constraint in the production setting was to extend the optimizer with as few rules as possible. This was for two obvious reasons. First the search space explored is directly proportional to number of rules rooted at each operator in query. Adding many rules can lead to a search space explosion and significant slow down in optimization time. Second, the production system collects telemetry and turns rules on and off on a per-customer basis. This is done to reduce compilation time and avoid performance regressions (occur commonly in cost-based optimizers due to costing inaccuracies). Adding many rules make such auto-tuning harder.

A natural question then arises as to how many such operator tree patterns are optimizable (referred to simply as patterns below), and how many new rules need to be added. We employed the synthesis heuristic on a slice of the daily production workload to collect different patterns. Within a week we discovered more than 100 such patterns for subqueries rooted on joins alone. The optimizer currently has only 20 join rules! Adding all such patterns as rules would clearly lead to state-space explosion, for both optimization and auto-tuning. Further, adding so many rules would lead to a large increase in the code base (each rule operator pair would require modifications to about 50 different files and 300 lines of new code).

Proposed solution

A careful investigation of the patterns revealed the following key observation. Many of the patterns that were optimized were logically similar, and could be implemented with minor modifications to the same super-operator. Consider for example the operator trees in Figure 3. The one on the left has one operator less and can be implemented by a super-operator with fewer conditional statements (we show the super-operators in section 3). The one in the middle filters data after the second *GroupBy*, and can be implemented with a different set of guards. The one on the right adds an additional column to differentiate the left and right rows in the unioned output. This can again be implemented with a minor change to the super-operator. It is easy to see that there are several more alternatives of the same pattern.

Based on this observation we propose four generic extensions to the existing query optimizer (1) a new abstract representation for operator trees matched by rules, that succinctly captures all patterns a single super-operator implements. (2) a new rule matching algorithm that efficiently searches for abstract operator trees, and checks if they can be implemented in a streaming manner (3) extensions to the post rule match pass to capture super-operator parameters from abstract edges (e.g. where were the select and project operators) (4) careful extensions to the physical operator interface to add parameterized super-operators, so that they implement all valid variants of the abstract operator tree.

With these extensions, 11 new substitution rules are sufficient to cover all the patterns discovered over a week (320 patterns used in 4000 queries). Each such substitution is guaranteed to eliminate several stage-breaking operators like *Join*, *Spool* and *UnionAll*, replacing them with an efficient streaming super-operator. We evaluated the optimization on several production queries and found the new rules provide significant improvements. On average across 20 production queries they simultaneously, speedup the query execution time by 1.5×, and reduce the cost of running queries (measured in terms of the cumulative task time across all stages) by one-third. The optimizations were achieved without impacting query optimization time. In fact, we found that whenever the rules fired they actually reduced overall query compilation time, as having fewer stages reduced the code-generation time. Finally we also found many more optimization opportunities than the synthesis-based system, as the integration into the query optimizer allowed it to exploit other rewrite rules that exist within the system.

The rest of the paper is organized as follows. In Section 2 we give an overview of the production query compilation pipeline and explain how a typical big-data optimizer decides to introduce stage breaks. In Section 3 we show several examples of super-operators and motivate the need for optimizer changes. In Section 4 we describe the core algorithmic extensions to the query optimizer. Section 5 discusses implementation details and challenges. Section 6 discusses related work and contrasts our techniques against them. Section 7 presents evaluation results and we conclude in Section 8. Below we summarize the key Contributions of this paper.

- We propose a powerful optimization that eliminates multiple stage-break inducing operators (*Join*, *GroupBy*, *Spool*, *UnionAll*), replacing them with a single streaming operator. We refer to such operators as super-operators.
- We introduce a new class of rules to enable such optimizations. They directly substitute abstract operator trees, containing optional operators, with parameterized super-operators. This way we add as few rules as possible.
- We add new algorithms to the big-data query optimizer to efficiently match such rules and enable optimization opportunities without affecting optimization time.
- We implement our optimizations in a production system with few well abstracted changes. Our parameterized super-operators are implemented in a modular and templatized way.
- We demonstrate that the super-operators significantly improve query performance.

2. PRELIMINARIES

This section provides an overview of the big-data query compilation process (see Figure 4).

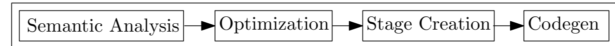

Figure 4: Optimizer Phases

Semantic analysis

The query script is first analyzed for errors and translated into a logical operator tree. The operator tree consists of nodes representing relational operators, and edges representing dependencies. Non-unary operators in the query (*Unions* and *Joins*) are decomposed into a sequence of binary operators. A special operator called *spool* is used to represent single producer multiple consumer sub-queries. Table 1 describes important operators and their attributes. Dependencies in the operator tree flow from bottom to top (children produce data consumed by parents). The logical tree for our running example is shown in Figure 5.

Query optimization

The logical operator tree is then optimized to generate a physical plan. This paper considers transformation based query optimizers as described in [11, 12]. Such optimizers have been used to build both industry standard query optimizers [3, 13] and academic prototypes [28]. The SCOPE optimizer also belongs to this family. Starting from the input tree, transformation rules are used to generate alternative query trees. There are two kinds of transformation rules. Exploration or logical transformation rules transform logical trees to produce new logical trees. Implementation or physical transformation rules transform logical trees into hybrid logical/physical trees. Implementation rules may introduce sorting and exchange operators where necessary. For example *merge joins* require the data to be partitioned and sorted on the key and the *merge join* implementation rule will add an exchange and a sort operator in the physical operator tree (Figure 5 shows the physical tree for our running example). All valid trees are maintained in a standard lookup table called a MEMO [27]. Each rule consists of a root operator, a source operator tree, and a target tree. The optimizer traverses the MEMO, and for each operator invokes all rules that are rooted on it. Many different algorithms for tree traversal and for cost based pruning have been discussed in literature but these are orthogonal to the key contributions of the paper, we refer to relevant aspects where needed (Section 5). We will take a more detailed look at the rule matching algorithm in Section 4.

Table 1: Operators and their attributes

Operator	Attribute description
Select ($\sigma_\phi(T)$)	Applies a filter predicate ϕ to each row of an input table T and outputs only rows that satisfy ϕ.
Project ($\pi_{A,p(cols(T))\rightarrow B}(T)$)	Retains a subset A of existing columns per row of T and creates some new columns B by applying functions p to columns of T
GroupBy ($\gamma_{A,f^*(cols(T)\rightarrow A)}(T)$)	A is a set of grouping columns. The operator applies commutative and associative aggregation functions ($f^*(cols(T))$) to rows in a group to produce an aggregated output.
(Equi)-Join ($\bowtie^{jt}{}_\psi(T_1, T_2)$)	An equi-join matches rows from T_1 with rows from T_2 on a conjunction of equality predicates. jt is a join type and can be any of *inner (i), leftOuter (lo), rightOuter (ro), leftSemi (ls), rightSemi (rs).*
UnionAll ($\bigcup_A(T_1, T_2)$)	Unions the rows from T_1 with the rows from T_2. After unioning it gives the columns new names A.
RestrRemap ($\rho_{cols(T)\rightarrow A}(T)$)	Renames the columns in T with new names A.
Spool ($S_n(T)$)	A single producer multiple consumer operator that duplicates the rows of T along n consumer edges.

Stage creation

Big data query optimizers [32, 4, 26] translate the chosen physical plan into a DAG of stages so that each stage runs in a data-parallel manner on many machines. The output data of a stage is materialized, and transferred between stages over the network via a shuffle. SCOPE breaks the physical plan into different stages at the following operators [32].

- **Exchange**: Exchanges re-partition and re-sort the data so that it satisfies the required properties of the parent operator. The operators that expect partitioned or sorted inputs are *GroupBy, OrderBy, Join, WindowedAggregation, Output and Reduce.* Such a re-ordering of data necessitates a stage break.
- **Join and Union**: *Joins* and *Unions* combine data from multiple sources. When data is produced by different stages, they introduce a separate stage.
- **Spool**: *Spools* produce data for multiple consumers. The output of a *Spool* operator is buffered until all consumers read data. As the consumers may consume data at different rates spools have unknown memory requirements. In the big-data setting, executors only have a limited amount of memory (a few GB) and would fail if this limit is exceeded. The optimizer therefore conservatively introduces a stage-break at a *Spool.*

Other big-data systems like HIVE and Spark also introduce stage-breaks in a similar manner and we discuss this in related work.

Figure 5 show the physical tree and stages generated for our running example. The physical plan has a *Spool*, two *GroupBy*'s and a *UnionAll.* This is split into 4 stages as shown in the stage plan. A stage-break in the plan implies that data needs to be shuffled between two stages. The resource cost and latency of a shuffle depends on size of the data and number of machines involved. In SCOPE, like

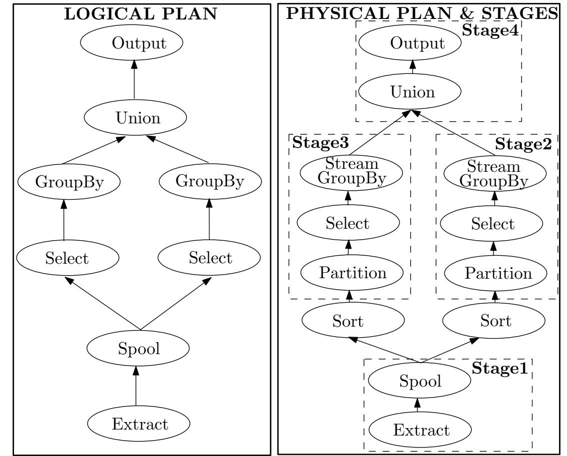

Figure 5: Logical and physical plan along with stages

in other massive deployments (Spark at Facebook for e.g. [30]), a shuffle sometimes dynamically introduces additional aggregation stages [21]. This is done to limit the number of machines that a single vertex reads from (fan-in) or writes to (fan-out) and limit the impact of failures on query execution. As we shall see later (figure 16) a query with this structure actually runs with 9 stages.

Code generation and run-time execution

Finally executable binaries are generated for each of the stages. The physical operators are implemented in a templatized form. These templates are then instantiated during code generation (SCOPE uses Microsoft's T4 template transformation toolkit [1] for this) to produce low level code. And this code is compiled down to stage binaries.

Efficient code-generation for SQL queries has been a subject of several papers [20, 17, 16, 8, 19, 25]. These compilers perform many low level optimizations, like eliminating unnecessary virtual function calls and redundant computations of expressions. However, high-level semantic optimizations are left to the query optimizer. See section 6 for a detailed discussion. The execution of the query is orchestrated by the big-data runtime engine, using standard resource management frameworks like YARN [7]. Vertices that execute a stage run within containers with a fixed amount of memory and CPU. The vertex fails if the memory limit is exceeded. The materialized output from each stage is managed by the runtime, and a shuffle service takes care of data transfers.

3. SUPER-OPERATORS AND ABSTRACT OPERATOR TREES

In this section we look at three example super-operators and the operator trees they implement. These are the super-operators that occur most frequently in practice, and together cover 40% of the patterns we have seen.

3.1 Abstract operator trees

As discussed in the previous section, the optimizer transforms operator trees through rewrite rules. We introduce the notion of abstract operator trees to compactly represent all operator trees that a single super-operator implements. An abstract operator tree uses abstract edges to represent a

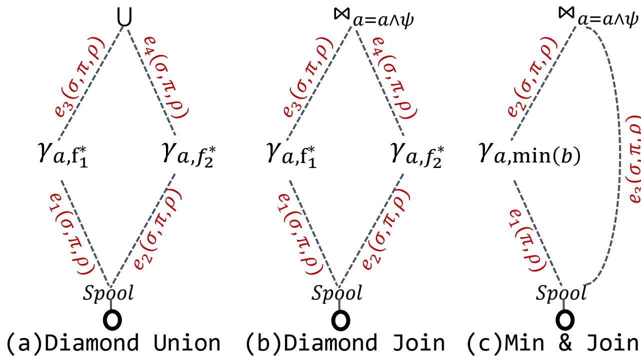

Figure 6: Abstract operator trees.

chain of zero or more optional operators between a pair of core operators. Each abstract edge is annotated with a set of unary operators, each of which can occur zero or more number of times and in any order between the child core operator and the parent core operator. We refer to the operators that form the nodes of an abstract tree as core operators and the operators on the edges as optional operators.

3.2 Super-operator examples

Figure 6 shows abstract operator trees that are implemented by our three example super-operators. We will examine some of the interesting (concrete) variants below and revisit the super-operators towards the end of each example.

Diamond UnionAll

Consider the simplest variant of Figure 6(a), one with no optional operators (concrete instantiation shown in Figure 7).

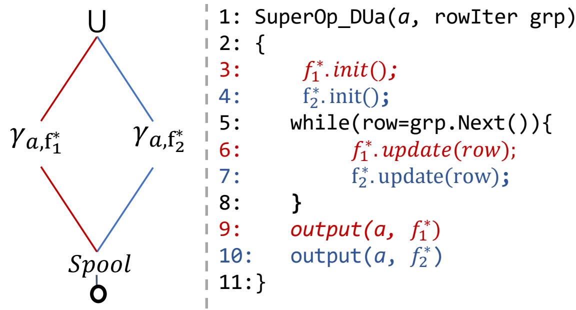

Figure 7: Diamond UnionAll Core

This query simply *Unions* the results of two different set of aggregates grouped by column a from the same input. We use the shorthand $f_i{}^*$ to represent one or more aggregations being applied to columns input to the *GroupBy*. For simplicity we also refer to the column names output by the aggregations as $f_i{}^*$. It is easy to see that the query is group-wise [6], i.e. its input can be partitioned on the group-by key a and each partition can be computed upon independently. Figure 7 also shows a super-operator that implements this query in a streaming manner. The super-operator takes as input a grouping key and an iterator over the rows belonging to the group. It initializes all aggregations on lines 3 and 4 (from both inputs to the *UnionAll*), and invokes the update function of the aggregations on every row (lines 6 and 7). This logic is similar to *PhyOp_ StreamGbAgg*, a standard physical operator that implements group by with aggregation. As the aggregations are commutative, the operator does not require the rows within a group to be sorted in any particular way. At the end (unlike *PhyOp_ StreamGbAgg*) the super-operator outputs two rows to simulate a *UnionAll*, one containing the aggregations from the left (line 9), and one containing aggregations from the right (line 10). It follows that the super-operator implements the query. This query produces a two stage plan once the super-operator optimization is enabled. Without the super-operator the query plan has four stages as we saw in Figure 5. As described in the previous section *Spool* introduces a stage break as its default implementation requires materialization of its output so that multiple consumers can process it, the two *GroupBy*'s separately consume the spool output in separate stages, and finally the *UnionAll* introduces a separate stage to combine the output of two different stages. Such a 4 stage plan is produced not only by SCOPE but by other state-of-the-art big-data query engines as well [4, 26]. Determining that the computation can be performed in 2 stages requires the optimizer to look at many operators together, as we propose.

Next lets consider a variant with *Select* operators along some of the abstract edges. Consider the concrete tree shown in Figure 8 which has a filter before the aggregation on the left, and a filter after the aggregation on the right.

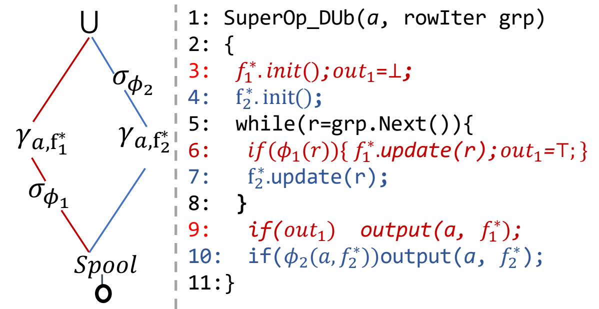

Figure 8: Diamond UnionAll with Select

This query can be implemented by a super-operator that has a similar structure but some of the statements are now guarded by conditions. First, the updates to the aggregates on the left (line 6) are guarded by the predicate ϕ_1 on the input row. Further an additional flag, out_1, is used to track if any row satisfies this predicate. And this flag guards the left output of the *UnionAll* (line 9). This guard is necessary to correctly implement the case where no element in the group satisfies predicate ϕ_1, and the left aggregation does not produce any output for that group. In summary lines 3,5,6,8 and 9 implement the left sub-query. The aggregations along the right path are computed as before. The final output (line 10) is now guarded by predicate ϕ_2. Its easy to see that lines 4, 5, 7, 8 and 10 implement the right sub-query. And hence the lines put together implement the entire query.

Now lets consider a variant with *Project*. Projections can either drop existing columns, introduce new constant columns or produce new columns by applying a function to existing columns. Consider the query shown in Figure 9.

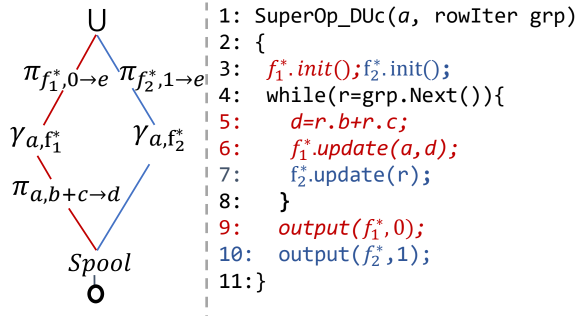

Figure 9: Diamond UnionAll with Project

The super-operator that implements this query now has additional expressions to compute the new column and perform aggregation on it (lines 5,6). It also modifies the output statements (lines 9,10), to drop projected columns, and introduce new constant columns as necessary.

```
1: List<edgeparams> p
2: SuperOp_DU(a,rowIter g){
3:  f1=init; out1=⊥;
4:  f2 =init; out2=⊥;
5:  while(r=g.Next()){
6:   if(<p[1].ϕ>(r)) { f1*.update(<p[1].π>(r)); out1=⊤; }
7:   if(<p[2].ϕ>(r)) { f2*.update(<p[2].π>(r)); out2=⊤; }
8:  }
9:  if(out1 ∧<p[3].ϕ>(r)) OUT(<p[3].π>(a,f1*));
10: if(out2 ∧<p[4].ϕ>(r)) OUT(<p[4].π>(a,f2*));
11:}

12: List<edgeparams> p, joinType jt
13: SuperOp_DJ(a,rowIter g){
14-19:repeat of 3-8 as DJ computes same aggregates
20: if(out1 ∧<p[3].ϕ>(a,f1*) ∧ out2 ∧<p[4].ϕ>(a,f2*))
21:    if(ψ) OUT(<p[3].π>(a,f1*), <p[4].π>(a,f2*));
22: if(jt=lo ∧ (out1 ∧ <p[3].ϕ>(a,f1*))∧ ¬(out2 ∧ <p[4].ϕ>(a,f2*))
23:    if(ψ) OUT(<p[3].π>(a,f1*),null);
24: if(jt=ro ∧ ¬(out1 ∧ <p[3].ϕ>(a,f1*))∧ (out2 ∧ <p[4].ϕ>(a,f2*))
25:    if(ψ) OUT(null,<p[4].π>(a,f2*));
26:}

27: List<edgeparams> p, joinType jt
28: SuperOp_MJ(a,sortedRowIter g, sortCol b){
29: first=⊤; lsOut=⊥;
30: while(r=g.Next()){
31:    if(first){min=b; first=⊥; }
32:    if(<p[3].ϕ>(r)) lsOut = ⊤;
33:    if(<p[2].ϕ>(a,min)∧<p[3].ϕ>(r) ∧ ¬(jt=ls))
34:        if(ψ) output(<p[2].π>(a,min),<p[3].π>(r));
35:    if(jt=ro ∧ ¬(<p[2].ϕ>(r) ∧ ψ) OUT(null,<p[3].π>(r));
36:    if(jt=lo ∧ ¬(<p[3].ϕ>(r)) OUT(<p[2].π>(a,min),null);
37: }
38: if(lsOut∧<p[2].ϕ>(a,min)) OUT(<p[2].π>(a,min));
39:}
```

Figure 10: Parameterized Super-Operators

The above examples show how the same super-operator can be extended to support different operator types along the abstract edges. We describe an algorithm to check whether a query tree is an instance of an abstract tree in Section 4. Figure 10 shows a parametric super-operator ($SuperOp_DU$) that implements all instantiations of the abstract tree shown in Figure 6(a). The super-operator is parameterized by a list of edge parameters, one per abstract edge. Each edge parameter ($p[i]$ for edge e_i) consists of (1) a summary predicate ϕ that summarizes all filtering that happens along the abstract edge and (2) a summary projection list π that, summarize all transformations that happen along an abstract edge, and directly relates the columns input to the parent operator with the output of the child. An algorithm to construct such edge parameters is described in Section 4.4. The parameters are used to introduce appropriate guards and expressions in the super-operator as motivated by the examples above.

Performance wise we expect this super-operator to be much more efficient than the original query, as it substitutes a *Spool*, a *UnionAll* and two *GroupBy*'s, each of which typically induces a stage-break, with a single operator.

Example : Diamond Join

The abstract tree in Figure 6(b) is similar to Figure 6(a) except that it performs a *Join* instead of a *UnionAll*. As shown in method $SuperOp_DJ$ of Figure 10 the super-operator for this is also similar except in the conditions for output. To support different types of joins, the super-operator is additionally parameterized by $joinType$. Note that both left and right sub-queries are grouped on a, and the join has an equality predicate on a (this is necessary for a tree to be a concrete instantiation of Figure 6(b)). This implies that each row from the left can join with at most one row from the right and vice versa. Therefore semi joins and inner joins behave the same way. They produce an output only if some input rows were aggregated on each side, and the results were not filtered out (lines 20-21). Note the results could be filtered out either by filters before the join ($e_3.\phi$ or $e_4.\phi$) or by ψ the additional conjunctive predicates of the equi-join. Outer joins produce an output row with *nulls* even if one of the sub-queries does not produce an output (lines 22-25).

Performance wise we expect this super-operator to be much more efficient than the original query as it substitutes multiple stage-break inducing operators (a *Spool*, a *Join* and two *GroupBy*) with a single streaming super-operator.

Example : Min and Join

Figure 11 shows a concrete variant of the third super-operator, one with no optional operators. The query, groups the elements of the input on column a, and computes the minimum value for column b. It then joins back with the input on column a, in effect adding a new column to every row of a group that contains the minimum value for column b. The super-operator that implements this query is shown in Figure 11. It requires data to be partitioned on a, and makes use of a sorted iterator that streams the elements within the group sorted on column b. The implementation exploits the fact that after sorting on b the first row has the minimum b value. Therefore, at line 6 of Figure 11 it assigns the first b value to min. It then appends min and outputs (line 7) all elements of the group in a single pass. We highlight that the sorting that the operator requires is only for elements within a partition, and can be done at a low performance cost. In the big-data setting, partitioning and sorting within a partition, is efficiently supported by the shuffle implementation.

Next lets consider a variant with filters along the edges that are input to the inner join, as shown in Figure 12. The super-operator that implements this has to only slightly be modified so that the output statement is guarded by a conjunction of the two predicates (line 7, Figure 12). More

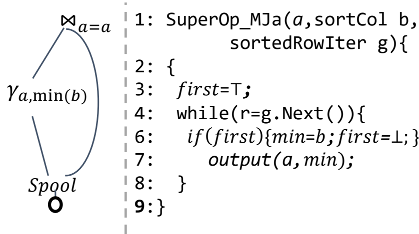

Figure 11: Min & Join Core

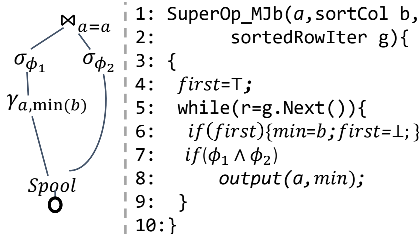

Figure 12: Min & Join with Select

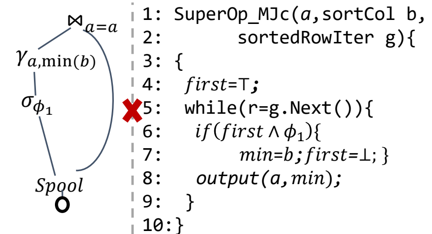

Figure 13: Non-streaming

interestingly turns out that the streaming super-operator cannot be extended to support a *Select* along the input edge to the aggregation, as shown in Figure 13. Sorting on b is not enough anymore, as the first row may be part of the output but may no longer be the min value satisfying the filter predicate ϕ_1. In the incorrect implementation shown in the figure the min value would be unassigned. A correct implementation would need to buffer inputs till a row that satisfies ϕ_1 is seen. As the buffer has a non-constant memory requirement it would violate the streamability property. We therefore constrain the abstract tree that this operator implements as shown in Figure 6(c) to not contain *select* along the abstract edge from input to the min aggregate. Replacing an operator tree with a single non-streaming operator introduces non-trivial trade-offs. In this paper we only focus on streaming super-operators, and defer discussion and evaluation of such operators to future work.

The complete super-operator (method *SuperOp_MJ* Figure 10) implements the abstract operator tree in Figure 6(c). It generalizes the examples above to deal with other types of joins. As the join is many to one, semi-joins induce additional complications. We skip a detailed discussion in the interest of space. This super-operator eliminates a *Join* and a *Spool*. Note that some simple variants of this pattern could also be written with windowed aggregations. Our optimizer uses known rewrite rules [33, 9] to decompose windowed aggregates to *GroupBy* and *Join* (as in the abstract tree) to explore additional implementation choices. Hence the same abstract tree would match even if the query were written differently. In terms of performance, the super-operator provides a specialized implementation of min and max windowed aggregations.

3.3 Characteristics of abstract operator trees

The goal of the proposed optimizations is to eliminate shuffles by substituting multiple shuffle inducing operators, with a single *streaming* super-operator. An operator is streaming if, given partitioned and sorted data, it can process all input rows while maintaining only a constant amount of state. Streaming operators are preferred in the scale-out big-data setting [32] as, they require bounded memory, and can be pipelined with other operators. Checking for streamability typically requires checks to be performed on grouping and join keys and on aggregation functions used.

The abstract operator trees for all the streaming super-operators that we have encountered so far (3 examples from this section and 8 others) share three common characteristics. First, they represent single-input single-output subqueries. That is, they have a single leaf operator and no out edges other than from the root operator. Second, shuffle inducing operators always appear as core operators. Attributes of such operators determine if the operator tree can be implemented by a streaming, single-input single-output, super-operator. Checking attributes of core operators (Section 4.3) therefore suffices to enable a substitution. In our experience we find abstract trees with 3-7 core operators composed of combinations of (*Join, UnionAll, GroupBy, OrderBy, Spool and Output*). The only shuffle inducing operators that are not covered by any of our super-operators are *windowed Aggregation* and (custom) *reducer*. Windowed aggregations, as discussed before, are often rewritten with *GroupBy* and *Join* using standard equivalences [33, 9], our optimizations therefore cover queries that perform windowed aggregation. Custom reducers use non-SQL code and optimizing such code in conjunction with other SQL operators is beyond the scope of this work.

Third, they contain other unary operators (that run in the same stage as their children) as annotations on the abstract edges. Attributes of these operators do not influence the validity of the substitution, but determine the control flow and the expressions computed within a super-operator. In our experience we find *Select*, *Project*, *RestrRemap* appear quite commonly in optimizable queries. Our implementation only supports them for now. *Process* (custom mappers) and *Top* are the only other operators we find in optimizable queries. *Process* like *Reduce* is hard to optimize, extending our algorithms to support *Top* is straight-forward.
We build on these observations to extend the query optimizer to support super-operators and their rules.

4. EXTENDING THE QUERY OPTIMIZER

The query optimizer traverses operators in the MEMO and applies rules until no new entries are added (Section 2). This section describes the existing rule matching algorithm and extensions needed for super-operators.

4.1 Current Rule matching algorithm

Each rule consists of a root operator, a source operator tree and a target tree. As shown in Figure 14 the rule matching algorithm has two steps, a light-weight pattern match on the operator names called AdvanceNext() and a detailed check which ensures that the operators in tree have the right attributes, called CheckAttributes() . AdvanceNext() matches the existing sub-tree in the MEMO with the source tree of the rule. AdvanceNext() traverses down the tree matching operators at every level until it matches all nodes in the source tree. In order to keep the check light-weight only operator names are matched. Once all operators are matched, AdvanceNext() succeeds and CheckAttributes() is executed. The exact checks in CheckAttributes() vary from rule to rule. For example, a rule that tries to push a *Select* below a *Join* would check

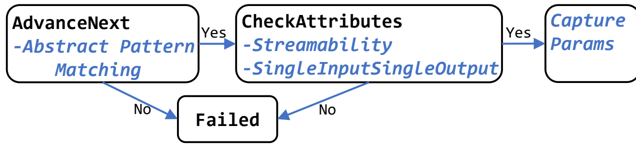

Figure 14: Current rule matching algorithm and extensions need for super-operator rules (blue)

if all the columns that are used in the selection predicate come from a single source of the join. If so it is safe to push the *Select* before the *Join*.

4.2 Rule matching with abstract trees

As described in Section 3, the source trees for super-operator rules can be compactly represented by an abstract operator tree. We extend the rule matching algorithm to match abstract operator trees as below. We modify AdvanceNext() to only check that the core operators occur in the right order. The checks start with child abstract edges of the root operator. Recall that an abstract edge is annotated with a set S of optional operators that can occur any number of times. As AdvanceNext() traverses an abstract edge it checks if the operator encountered belongs to S. If so it proceeds to its child. This process is repeated until an operator that is not in S is encountered[1]. If this is not the core operator at the source of a abstract edge then AdvanceNext() fails. Else it continues traversing the next abstract edge in a similar manner. The process is repeated until all core operators are matched and then the rule matching advances to CheckAttributes().

During CheckAttributes() we perform two checks that are common to all super-operator rules. First we check that the matched operator tree is single-input single-output. As our super-operators are single-input single-output we need to perform this check to ensure that the substitution is valid. Checking this property requires additional checks for multi-input (*Join*, *UnionAll*) and multi-output (*Spool*) operators. We ensure that for non-root multi-output operators all their parents are part of the matched sub-tree and for non-leaf multi-input operators all their children are part of the matched sub-tree. Further we check that the matched query has a single leaf operator (AdvanceNext() only checks if the operator names match those in the source tree but not if they are the same instance). Second we check that the operator tree can be implemented in a streaming fashion using a Streamability Check. The exact check varies from rule to rule. Below we describe the checks for our 3 examples.

4.3 Streamability Checks

We describe the streamability checks for our three super-operators from Section 3. The goal is to establish checks that are sufficient to ensure that the super-operators implement the operator tree in a streaming manner.

The streamability check for the first two operators (Algorithms 1 and 2) build on the observation that a single

[1]The algorithm assumes that the child operator of an abstract edge is not in S. This is a natural assumption for super-operator optimizations (Section 3.3). Extending the algorithm to handle overlapping operators is not very hard.

Algorithm 1 Diamond-UnionAll Op. Streamability Check

```
1: procedure CheckStreamability(Γ1, e1, Γ2, e2)
2:     set1 = TraverseDown(Γ1.keys, e1)
3:     set2 = TraverseDown(Γ2.keys, e2)
4:     if set1 ≠ ∅ ∧ set1 == set2 then
5:         return true
6:     else
7:         return false
```

Algorithm 2 Diamond-Join Op. Streamability Check

```
1: procedure CheckStreamability(spool, Γ1, Γ2, ⋈,
   e1, e2, e3, e4)
2:     set1 = TraverseDown(Γ1.keys, e1)
3:     set2 = TraverseDown(Γ2.keys, e2)
4:     if ¬(set1 ≠ ∅ ∧ set1 == set2) then
5:         return false
6:     set3 = TraverseUp(Γ1.keys, e3)
7:     set4 = TraverseUp(Γ2.keys, e4)
8:     if ¬(set3 ≠ ∅∧ set3.size = set4.size) then
9:         return false
10:    pred = ⊤
11:    for i ∈ (1...set3.size) do
12:        pred = pred∧ (set3[i] == set4[i])
13:    if ⋈.eqPred ⟹ pred then
14:        return true        ▷ ⋈ keys are superset of Γ keys
15:    else
16:        return false
```

GroupBy with aggregations can be implemented in a streaming fashion if the input is partitioned on the grouping keys. As a natural extension two *GroupBy*'s on two different paths can be simultaneously computed in a streaming fashion if the grouping keys on both sides are direct or renamed versions of the same input columns. **Diamond UnionAll** simply unions the results of two paths so we check this condition in line 4 of Algorithm 1. Optional renaming operators along the edges from the input to the aggregations can rename the grouping keys. To check if the keys are indeed the same such renamings need to be undone before performing the check. Helper routine TraverseDown (Algorithm 4) is used to undo such mappings. Given a set of columns and an edge e, it reverses the renamings done by the *map* functions in *RestrRemap* operators. Lines 2-3 of algorithm 1 use this function to remap grouping keys along the left and right path to the input before applying the check at line 4.

Diamond Join, needs an additional check to ensure that the join keys are a super set of the grouping keys. This check is somewhat involved. To check this, first the grouping keys are mapped to the join (lines 6,7 of Algorithm 2) using a TraverseUp routine (not shown as it is basically the inverse of TraverseDown). Then we construct a predicate that is a conjunction of equalities between one key from the left and one from the right (lines 10-12). Finally we check if the equi-joins conjunctive predicate implies this equality predicate (line 13). This would mean that the equi-join checks for equality between all pairs of grouping keys. If this is true then the join can be simulated in a streaming manner (as in Section 3) by concatenating the aggregated values from the two sides and outputting them under a guard.

Min and Join does not contain a *GroupBy* on the right side. For such abstract trees the output is linear in the input

Algorithm 3 Min & Join Op. Streamability Check

1: **procedure** CheckStreamability(spool, Γ, $\bowtie$, e_1, e_2, e_3)
2: set1 = TraverseDown(Γ.Keys, e_1)
3: set2 = TraverseUp(Γ.Keys, e_2)
4: set3 = TraverseUp(set1, e_3)
5: **if** $\neg$(set2 $\neq \emptyset \wedge$ set2.size = set3.size) **then**
6: **return** false
7: $pred = \top$
8: **for** $i \in (1...set2.size)$ **do**
9: $pred = pred \wedge$ (set2[i] == set3[i])
10: **if** $\neg(\bowtie.eqPred \implies pred)$ **then**
11: **return** false
12: **if** Γ.aggregationList.Count() ==1 **then**
13: agg = aggregationList[0]
14: **if** (agg is $min(b)$)$\vee$ (agg is $max(b)$) **then**
15: set4 = undoMapping(b,undoMap1)
16: **if** $set4 \neq \emptyset \wedge \|set4\| == 1$ **then**
17: **return** true
18: **return** false

Algorithm 4 TraverseDown Procedure

1: **procedure** TraverseDown(cols, e)
2: $sources = cols$
3: $parentToChildOpList = e.ops.Reverse()$
4: **for** $curr \in parentToChildOpList$ **do**
5: **if** $curr$ is RestrRemap **then**
6: **for** $(a \rightarrow b) \in curr.maps$ **do**
7: **if** $b \in sources$ **then**
8: $sources.Remove(b)$
9: $sources.Add(a)$
10: **return** $sources$

only if the grouping key on the left is also used to perform a self-join. Once again this check is performed in a manner similar to **Diamond Join** (lines 2-11 Algorithm 3). For the query to be streaming it further needs to be able to perform the aggregation without going over all rows of the group (if aggregation requires one loop, then adding the aggregation to each row in the group would need a second pass over the rows making it non-streaming). This is only possible for Min and Max which can be computed if the input is sorted on the aggregation column. Lines 12-17 in the algorithm check that there is exactly one aggregation applied and that it performs a Min or a Max on a single input column.

4.4 Capture Parameters

In addition to AdvanceNext() and CheckAttributes() we add an additional step, CaptureParameters() , that is specific to the new rules. As we alluded to in Section 3 each super operator is parameterized with a list of edge parameters and a list of core operator parameters. This step captures such parameters. Algorithm 5 describes an algorithm that captures edge parameters for an abstract edge annotated with *Select*, *Project* and/or *RestrRemap*. We are in the process of adding support for *Top* the only other relevant optional operator (Section 3.3). Each edge parameter needs to contain enough information to relate the input columns of a core operator with the output columns of its child core operator. The algorithm maintains an expression table with one entry per output column (initialized to identity) and a summary predicate (initialized to true) to collect this information.

Algorithm 5 CaptureParameters

1: **procedure** CaptureParams(coreOp1, coreOp2,e)
2: $\forall_{col \in coreOp2.cols}\ projectionMap.Add((col \rightarrow col))$
3: pred = $\top$
4: curr = coreOp2.Child
5: **while** curr != coreOp1 **do**
6: **if** curr is RestrRemap **then**
7: **for** $(a \rightarrow b) \in curr.maps$ **do**
8: **for** $(col \rightarrow Expr) \in projectionMap$ **do**
9: **if** Expr.Contains(b) **then**
10: Expr.Substitute(b,a)
11: **if** pred.Contains(b) **then**
12: pred.Substitute(b,a)
13: **if** curr is Project **then**
14: **for** $(srcExpr \rightarrow b) \in curr.newCols$ **do**
15: **for** $(col \rightarrow Expr) \in projectionMap$ **do**
16: **if** Expr.Contains(b) **then**
17: Expr.Substitute(b,srcExpr)
18: **if** pred.Contains(b) **then**
19: pred.Substitute(b,srcExpr)
20: **if** curr is Select **then**
21: pred = pred $\wedge$ Select.predicate
22: curr = curr.Child
23: **return** (projectionMap,pred)

It walks down the abstract edge (from parent to child) and updates these entries as shown in Algorithm 5. Recall that each edge can have optional operators occuring any number of times in any order. On coming across *RestrRemap* (lines 6-12) every renaming is undone. That is, every occurrence of the new name is replaced with the old name in the summary predicate (lines 8-10) and in all expressions in the projection list (11-12). The algorithm makes use of an expression library to match variables (*Contains*) and perform alpha conversion (*Subst*) [2]. On a *project*, only newly added columns (*newCols*) need to be processed. Columns that were removed could not have been used in parent expressions or predicates. The newly added columns are processed similar to renames (lines 15-19), except that now the new column is replaced by an expression, growing the expression trees and the summary predicate. Finally on a *select* the summary predicate is updated by conjoining it with the select predicate (line 21). This process is repeated for every abstract edge. For example, for an edge with the sequence of operators $\sigma_{a+d>100}.\pi_{a,b+c\rightarrow d}$ the edge parameters would be $([a \rightarrow a, d \rightarrow b+c], a+b+c > 100)$.

In addition to this CaptureParameters() also analyzes the core operators to capture the type of the join and the column on which aggregations are performed (the column on which *min* is performed and the input column that it is renamed from) to construct a node parameter list. This is fairly trivial and we skip details here.

4.5 Costing super-operators

As the new super-operators introduced are streaming in nature they do not have any associated I/O cost (the plan may have sort and exchange operators before the super-operator and the plans cost will account for them). The CPU cost of the super-operator can be derived by relating

them to the cost of similar streaming operators. For example, by summing the CPU cost of the underlying grouping aggregations for **Diamond UnionAll** and **Diamond Join**. Note that the cost of the individual grouping aggregations would account for selectivities of intermediate filters.

5. IMPLEMENTATION

We cover some salient aspects of our implementation.

5.1 Physical operator and code generation

As SQL is a very rich language supporting many different types and type specific semantics (e.g. dealing with *null* during aggregation), re-implementing the core logic for our super-operators is bound to be error prone. We therefore implement them in a modular and templatized way.

- We reuse core methods used in existing operators to handle types and apply functions (built-in functions, aggregations, type conversion) on types. For example, the super-operator for our first two examples is similar to a streaming group-by implementation. This operator already has methods to determine when to trigger aggregation, how to perform any standard aggregation on any supported type and when to produce outputs.
- Super-operators need to evaluate predicates and expressions before invoking core methods. Big-data languages like SCOPE support non-SQL predicates and expressions, often as part of row-wise operations like *select* and *project*. The existing implementation of such operators already uses templatized code that has holes in code filled in during code generation. Our physical operator implementations make use of the same strategy. They leave holes which are filled based on super-operator parameters during code generation. A low level compiler that supports the language of the expressions and predicates is used to compile down the final binaries.
- Lastly our operators do require some super-operator specific code, for example for outputting rows by combining results from different core-operators. The code we added constitutes a very small fraction of the operator code and is carefully reviewed/tested.
- Super-operators sometimes share some common functionality. For example **Diamond UnionAll** and **Diamond Join** only differ in the output statements. The per-row computation they perform are the same. We therefore even implement our physical operators in a modular fashion, abstracting away common functionality into separate functions, and reusing them across super-operators.

5.2 Implementation summary and status

A major constraint while adding super-operators to production big data optimizer was to make few well abstracted changes to the code. In order to keep code changes to the minimum and to provide the right auto-tuning knobs we had to keep the number of new rules we add to a small number. To adhere to these constraints we developed a new core concept of abstract pattern matching described earlier.

So far we have added 3000 lines of new code, this includes 2000 lines to implement the core algorithms from the previous section and the rest for implementing super-operator specific logic (streamability checks and physical operators).

6. DISCUSSION & RELATED WORK

The optimizations proposed in this paper merge computations performed across multiple stages of a big-data query into a single stage, and produce efficient code (constant memory, single tight loop) for that stage. In this section we discuss two specific lines of work that address parts of this problem, but do not achieve the dual benefits above.

The first line of work tries to reduce the number of stages needed to execute a query. The production optimizer we use, already explores rules that eliminate exchange and sort operators [31] from a linear chain of operators. A related optimization that is part of the HIVE query optimizer [26, 18], tries to put multiple key based operators (*Join* and *GroupBy*) in the same stage. They look for correlations among operators and their children, and combine them into a single stage. Such an optimization only moves around stage boundaries, but does not eliminate non-streaming *Join* and *Spool* operators, nor does it produce a single tight loop implementation of multiple operators. Also their conditions are quite restrictive and different from ours (they do not look for streamability). They do not cover the patterns we optimize. We observe that the query optimizer used by Spark classifies transformations as wide or narrow, and introduces stage breaks at every wide operator. While they use a different terminology they end up introducing stage breaks as described in Section 2.

This paper proposes optimizations that go beyond basic exchange elimination or re-arranging of stage boundaries. They eliminate expensive *Join*, *Spool* and *UnionAll* operators altogether. Further, as the optimizations are integrated as rewrite rules, they compose with other existing rules.

A second line of work relates to advancements in code generation [20, 17, 16, 8, 19, 25]. These frameworks target low level inefficiencies common to data-analytics runtimes, including virtual call overheads incurred during query evaluation and computation of redundant expressions across multiple operators. This style of compilation happens after query optimization, and only looks for low level peep-hole style optimizations. In the big-data setting, Spark uses a code-generator derived from HyPer [20]. However it only looks for optimizations within a single stage [4, 8], and cannot perform cross stage optimizations as proposed here. The SCOPE query optimizer performs simple cross-stage peephole optimizations. It pushes down projections and filters [14] from scripts that also use non-SQL user code, and extends early aggregation [23] to user defined aggregates. The proposed optimizations are more powerful, they eliminate stages altogether by merging computation across stages, and produce efficient code.
In summary, by introducing non-SQL logical operators we enable powerful optimizations that current big-data query compilers (where optimizer deals with SQL operators and code generator produces per stage code) do not consider.

Note that while we focus on big-data settings in this paper, super-operator optimizations apply to other query engines as well. Even on a single machine, the optimizations would reduce the number of operators, reduce contention and eliminate unnecessary buffering. They would produce efficient code-fragments that, to the best of our knowledge, none of the existing code-generators do today.

Future Directions We would like to conclude with a discussion on potential ways to generalize the ideas presented here to a broader class of queries. It would be interesting to study if such optimizations can be applied to multi-output

queries, as targetted by recent code generators [25], or more broadly for multi-query optimization. A related problem is one of composing super-operators together via additional super-operator rewrite rules. From our experience we have seen instances where certain super-operators compose with other super-operators (e.g. two Diamond UnionAll). We plan to work on these problems in future.

7. EVALUATION

Big data systems like SCOPE, Hive and Spark are designed to run on large clusters containing hundreds to thousands of machines. SCOPE is used extensively within Microsoft for running business critical analytical queries often processing tens to hundreds of terabytes of data.

7.1 Experimental Setup

We evaluate our enhancements on a production cluster of more than 50,000 commodity machines. Machines come from different generations of the intel family but typically have 16-24 cores, 128 GB of RAM, 4 HDDs and 4 SSDs. We were constrained to use at-most 1000 cores at a time.

We pick important production queries for evaluation. We classify a query as important if it runs multiple times a week (on different data) and is run by important customers (identified based on the group name, prod vs. test for example). To isolate the benefits from our optimization we pick 20 queries where more than 70% of the cost of running the query is spent on sub-queries that we optimize.

We begin with a detailed look at a production query and how it benefits from our optimizations. We then report results on all evaluated queries.

7.2 Super-operators in action

Figure 15 shows a query that analyses mailbox storage logs (2.8TB) to identify watermark events. It assigns the timestamp of the last event (of any type) as a high watermark and the latest of one of few events as a low watermark. This is done through a diamond union pattern. The rest of the query (not shown) does further time series analysis on the result of this sub-query.

Figure 16 compares query plans with and without super-operators. We begin by explaining default plan (Figure 16(a)).

- The input file is first read and partitioned on *MailboxGuid* in stage 1. Stage 2 performs a partial aggregation at the pod level. Note that this stage is dynamically introduced at runtime to scale shuffle [32, 21]. Its output is spooled to two consumers (1.5 TB to Stage 3 and 2.8 TB to Stage 4).
- Stages 3,5,7 and 4,6,8 perform similar computation. Stage 3(4) applies a filter, sorts and partitions the data for stage 7(8). Once again the runtime dynamically introduces an intermediate aggregation stages (5 and 6) to scale shuffle. Stage 7(8) merges the corresponding partitions from stage 3(4) and 5(6) and performs final aggregations.
- Stage 9 unions 73.6 GB of data from Stages 7 and 8.

Overall the baseline plan shuffles 15TB of data. It runs in 32 minutes and its cost in terms of total compute time (cumulative across all vertices of all stages) is 88 hours. The optimized query (Figure 16(b)) has only 4 stages.

- The first two stages are similar to the baseline. They partition the input on the grouping column. In this case the partitioning is introduced to satisfy the required properties of our super-operator.
- Note that the optimized plan has eliminated the spool altogether. It directly performs a single intermediate aggregate in Stage 3 (as opposed to 2 in baseline).
- Stage 4 merges corresponding partitions and invokes our super-operator. The super-operator computes all necessary aggregates and directly produces the unioned output.

The optimized plan runs in 18 minutes (1.8x speedup) and its cost is only 43 hours (50% cost saving).

A surprising observation (not shown in figure) is that the optimization resulted in an overall reduction of 10% in compilation time. Turns out that due to the elimination of several stages the optimized query took lesser time for code generation. We observed similar reductions for other queries too. For queries where our optimizations did not apply there was no significant difference in compilation time.

Table 2: Benchmark queries. The table shows operator trees for the optimized sub-query. Operators are suffixed with number of columns used, core operators separated by |, and execution statistics before and after optimization.

	op	reduction in		serialized sub-query tree
		stages	shuffleTB	(post-order traveral)
1	DU	$9 \rightarrow 4$	$26 \rightarrow 15$	$(\sigma_2\pi_4 \vert \gamma_2 \vert \sigma_1, \sigma_6 \vert \gamma_2 \vert \pi_1\rho_1)\cup$
2	DJ	$9 \rightarrow 4$	$20 \rightarrow 11$	$(\rho_2\pi_1 \vert \gamma_4 \vert \sigma_2, \rho_1 \vert \gamma_4 \vert \pi\sigma_1) \bowtie_1$
3	DU	$11 \rightarrow 5$	$18 \rightarrow 10$	$(\pi_1 \vert \gamma_5 \vert \sigma_3, \pi_1 \vert \gamma_5 \vert \sigma_1)\cup$
4	DU	$10 \rightarrow 5$	$15 \rightarrow 8$	$(\rho_1\sigma_1\pi_1 \vert \gamma_4 \vert, \rho_1 \vert \gamma_4 \vert)\cup$
5	DJ	$10 \rightarrow 5$	$14 \rightarrow 8$	$(\sigma_2\pi_2 \vert \gamma_4 \vert \sigma_1, \vert \gamma_5 \vert \pi_2) \bowtie_4$
6	DU	$9 \rightarrow 4$	$6.5 \rightarrow 4$	$(\rho_1 \vert \gamma_5 \vert \sigma_1, \sigma_1 \vert \gamma_5 \vert)\cup$
7	DJ	$8 \rightarrow 4$	$7 \rightarrow 4.5$	$(\pi_1 \vert \gamma_2 \vert, \vert \gamma_5 \vert \pi_1\sigma_3) \bowtie_4$
8	DJ	$11 \rightarrow 6$	$5 \rightarrow 3$	$(\sigma_1 \vert \gamma_3 \vert \rho_1, \sigma_2 \vert \gamma_2 \vert) \bowtie_1$
9	DU	$8 \rightarrow 4$	$5.5 \rightarrow 3$	$(\rho_2 \vert \gamma_7 \vert \sigma_5, \rho_1 \vert \gamma_7 \vert)\cup$
10	MJ	$9 \rightarrow 5$	$5 \rightarrow 3$	$(\rho_1\pi_1 \vert \gamma_2 \vert, \sigma_1\rho_1) \bowtie_1$
11	DJ	$12 \rightarrow 6$	$3 \rightarrow 1.5$	$(\rho_1 \vert \gamma_4 \vert \sigma_1, \vert \gamma_4 \vert \pi_2)\cup$
12	DJ	$8 \rightarrow 4$	$6 \rightarrow 4.5$	$(\sigma_1 \vert \gamma_2 \vert \rho_1, \sigma_2 \vert \gamma_2 \vert) \bowtie_2$
13	DJ	$6 \rightarrow 3$	$3 \rightarrow 2.5$	$(\pi_2\sigma_1 \vert \gamma_4 \vert \pi_2, \sigma_3 \vert \gamma_3 \vert \rho_2) \bowtie_3$
14	DU	$6 \rightarrow 3$	$3.1 \rightarrow 2$	$(\rho_1\sigma_1\rho_2 \vert \gamma_5 \vert, \sigma_3 \vert \gamma_5 \vert)\cup$
15	DU	$7 \rightarrow 3$	$2.7 \rightarrow 2$	$(\pi_1\sigma_1 \vert \gamma_6 \vert, \sigma_1 \vert \gamma_5 \vert \pi_2)\cup$
16	MJ	$7 \rightarrow 4$	$2.5 \rightarrow 2$	$(\sigma_5 \vert \gamma_6 \vert \sigma_1, \sigma_2\pi_1) \bowtie_5$
17	DJ	$8 \rightarrow 4$	$2 \rightarrow 1$	$(\vert \gamma_4 \vert, \sigma_1 \vert \gamma_4 \vert \pi_1\sigma\pi_2) \bowtie_4$
18	DU	$8 \rightarrow 4$	$2 \rightarrow 1.5$	$(\rho_1\pi_1\sigma_1 \vert \gamma_4 \vert \rho_1, \sigma_5 \vert \gamma_4 \vert)\cup$
19	MJ	$6 \rightarrow 3$	$2 \rightarrow 1.7$	$(\pi_1 \vert \gamma_2 \vert \sigma_1, \sigma_4) \bowtie_1$
20	DU	$6 \rightarrow 4$	$1.4 \rightarrow 1$	$(\sigma_6 \vert \gamma_6 \vert \rho_1\pi_1\sigma_4\pi_2, \sigma_1 \vert \gamma_7 \vert \pi_1)\cup$

7.3 Operator trees and execution plans

Table 2 reports the operators trees for the benchmark queries. Post order traversals of the trees are shown with core operators separated by '|'. The table shows evidence that the same abstract tree indeed gets instantiated in many different ways, validating the need for our optimizer extensions (the new tree representation, the new rule matching algorithms and the addition of parameterized super-operators). The abstract edges contain various operator combinations with up to 4 operators on an abstract edge and with the same operator sometimes appearing multiple times (queries 14,17,20). The table also summarizes execution plans, it shows that the optimization usually eliminates half the stages and shuffles 10-50% less data.

7.4 Cost and latency improvements

Figure 17 reports the fraction of cost needed to run super-operator optimized queries. The queries are sorted in de-

```
(SELECT ARGMAX(TS, EventId) AS EventId,
  ARGMAX(TS, ActivityId) AS ActivityId,
  ARGMAX(TS, Simplified(EventId)) AS SEvent,
  ARGMAX(TS, MachineName) AS MachineName,
   MailboxGuid AS MailboxGuidString,
  COUNT(*) AS cnt
FROM log
WHERE EventName IN ("[EventName1]",
              "[EventName2]","[EventName3]")
GROUP BY MailboxGuidString)
UNION ALL
(SELECT ARGMAX(TS, EventId) AS EventId,
  ARGMAX(TS, ActivityId) AS ActivityId,
  FIRST("High") AS SEvent,
  ARGMAX(TS, MachineName) AS MachineName,
   MailboxGuid AS MailboxGuidString,
  COUNT(*) AS cnt
FROM log
GROUP BY MailboxGuidString);
```

Figure 15: Sample Query

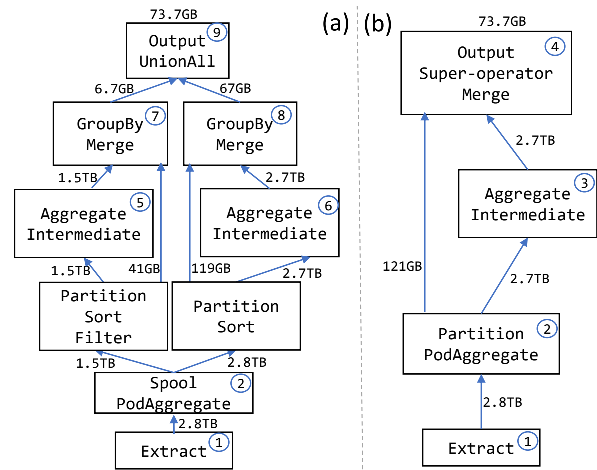

Figure 16: Query execution for sample query (a) before and (b) after super-operator optimization

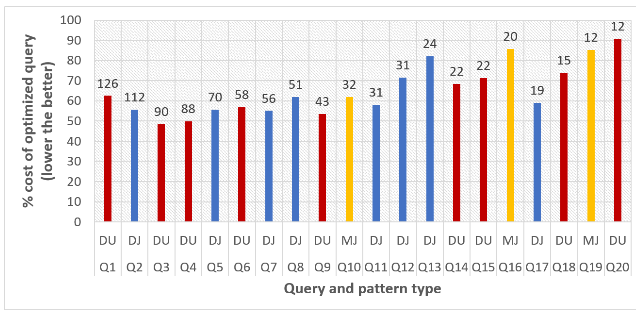

Figure 17: Cost savings. The plot shows the relative cost of optimized query as compared to the baseline. Each bar also shows the absolute cost in compute hours for the baseline query.

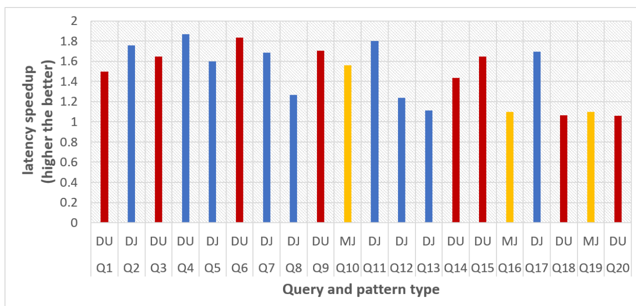

Figure 18: Latency speedup over the baseline.

creasing order of total cost of running the baseline query (cost in compute hours reported as a label on top of each bar). Each bar reports the % cost to run the query with a super-operator, relative to baseline. For the example query discussed earlier (Q4) the bar is at 50% and is labeled 88.

As expected, the reduction in cost is directly correlated to the amount of shuffle eliminated. A deeper look reveals that the reason for the differences in benefit do not so much dependent on the exact pattern eliminated but on how early data was filtered out. In particular if the data is filtered out before the aggregation, the lesser is the cost of the aggregation and the subsequent join/union operators, and hence the lower is the saving. Queries 2-3,7,9,11 do not have any filters before the aggregation. While queries 4-6,17 filter out data before the aggregation only along one path. All these queries save at-least 45% cost. A correlated effect is that queries with high costs have higher savings. The input data to most of our queries is in the range 1-4TB (except Q1 which has a 10TB input) and hence their relative cost depends on how much data is shuffled. As a consequence queries with low cost are also the ones with the least savings but we save significantly on costly queries.

Figure 18 reports the speedup measured as the ratio of the latency of the query before and after the optimization. The latency trends more or less follow the cost trends. The latency benefits are sometimes (Queries 3,8,18) lower than the cost benefits as the baseline exploits more machines to run some stages in parallel. On the other hand the latency benefits are higher in queries 6,11,15 as they have a few straggling tasks that delay the overall execution. In general as the baseline uses more tasks it is more likely to suffer from straggler effects. Overall the average speedup is 1.5×.

In summary we illustrate that by extending the optimizer to incorporate super-operators we produce more compact plans that simultaneously benefit both cost and latency. As a consequence the optimized plans use fewer tasks and can make more efficient use of the available parallelism.

8. CONCLUSIONS

In this paper we propose extensions required to transformation based optimizer like SCOPE to incorporate streaming super-operators. We answer the question of how to add minimum number of rules to the query optimizer for doing these transformations – required for code maintainability and efficiency – using novel concepts viz. abstract pattern matching and parameterized super-operators.

Acknowledgements

We would like to thank Marc Friedman, Max Reinsel, Sunny Gakhar, Shi Qiao and Clemens Szyperski from the SCOPE

team for reviewing our code and providing design feedback.

9. REFERENCES

[1] Code Generation and T4 Text Templates. https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates, 2019.

[2] Expression evaluation and lambdaCalculus. https://en.wikipedia.org/wiki/Lambda_calculus, 2019.

[3] SQL Server. https://www.microsoft.com/en-us/sql-server/sql-server-2019, 2019.

[4] 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 *SIGMOD*, pages 1383–1394, 2015.

[5] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 2008.

[6] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise processing of relational queries. In *VLDB*, pages 476–485, 1997.

[7] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, pages 177–192, 2019.

[8] 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.

[9] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[11] Goetz Graefe. The cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[12] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[13] Tandem Database Group. Nonstop sql: A distributed, high-performance, high-availability implementation of sql. In *High Performance Transaction Systems*, 1989.

[14] 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.

[15] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.

[16] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10), 2014.

[17] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE 2010*, pages 613–624, 2010.

[18] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, pages 25–36, 2011.

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

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

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

[22] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure data lake store: A hyperscale distributed file service for big data analytics. In *SIGMOD*, pages 51–63, 2017.

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

[24] Matthias Schlaipfer, Kaushik Rajan, Akash Lal, and Malavika Samak. Optimizing big-data queries using program synthesis. In *SOSP*, pages 631–646, 2017.

[25] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and XuanLong Nguyen. A layered aggregate engine for analytics workloads. In *SIGMOD*, pages 1642–1659, 2019.

[26] 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. *PVLDB*, pages 1626–1629, 2009.

[27] Florian Waas et al. Counting, enumerating, and sampling of execution plans in a cost-based query optimizer. In *ACM SIGMOD Record*, 2000.

[28] Yongwen Xu. Efficiency in the columbia database query optimizer. Master's thesis, Portland State University, 1998.

[29] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 2010.

[30] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. Riffle: Optimized shuffle service for large-scale data analytics. In *EuroSys*, pages 43:1–43:15, 2018.

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

[32] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. Scope: parallel databases meet mapreduce. In *PVLDB*, volume 21, pages 611–636, 2012.

[33] Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, and Kwai Wong. Winmagic: Subquery elimination using window aggregation. In *SIGMOD*, pages 652–656, 2003.