Exploiting Common Subexpressions for Cloud Query Processing

Yasin N. Silva [#], Per-Ake Larson [*], Jingren Zhou [+]

[#]*Arizona State University*
Glendale, AZ 85306, USA
ysilva@asu.edu

[*]*Microsoft Research*
Redmond, WA 98052, USA
palarson@microsoft.com

[+]*Microsoft Corporation*
Redmond, WA 98052, USA
jrzhou@microsoft.com

***Abstract*—Many companies now routinely run massive data analysis jobs – expressed in some scripting language – on large clusters of low-end servers. Many analysis scripts are complex and contain common subexpressions, that is, intermediate results that are subsequently joined and aggregated in multiple different ways. Applying conventional optimization techniques to such scripts will produce plans that execute a common subexpression multiple times, once for each consumer, which is clearly wasteful. Moreover, different consumers may have different physical requirements on the result: one consumer may want it partitioned on a column A and another one partitioned on column B. To find a truly optimal plan, the optimizer must trade off such conflicting requirements in a cost-based manner. In this paper we show how to extend a Cascade-style optimizer to correctly optimize scripts containing common subexpression. The approach has been prototyped in SCOPE, Microsoft's system for massive data analysis. Experimental analysis of both simple and large real-world scripts shows that the extended optimizer produces plans with 21 to 57% lower estimated costs.**

I. Introduction

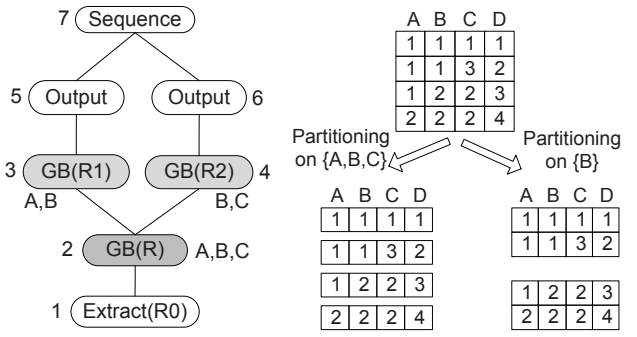
The analysis of massive amounts of data is a routine activity in many commercial and academic organizations. Internet companies, for instance, collect large amounts of data such as content produced by web crawlers, service logs and click streams. Analyzing these data sets may require processing tens or hundreds of terabytes of data. To perform this task, many companies rely on highly distributed software systems running on large clusters of commodity machines. Important examples of such systems are: Google's File System [1], MapReduce [2], [3], Bigtable [4], Apache Hadoop [5], Microsoft's Dryad [6], and SCOPE/Cosmos [7], [8]. The implementation of these systems presents multiple challenges and possibilities for the optimization of the queries or scripts used for analysis, here called cloud queries.

SCOPE [7], [8] is a SQL-like scripting language used within Microsoft for massive-scale data analysis. Thousands of SCOPE jobs run daily in Microsoft's data centers, processing petabytes of data and utilizing thousands of machines. Many SCOPE scripts are large and contain multiple queries. These queries can have many common subexpressions due to the nature of the analysis performed. For instance, many scripts first extract data from one or more input files and perform some initial aggregations. An aggregated result is often used in several places in the script where it might be further aggregated or joined. A conventional optimizer that does not deal with common subexpressions will produce a plan where the expressions are evaluated multiple times. In this paper we consider the problem of exploiting common subexpressions to optimize SCOPE scripts. Our framework allows the identification of common subexpressions and enables the generation of query plans where common subexpressions are executed once and their results used by multiple consumers. The selection of the best plan is performed in a cost-based fashion.

The SCOPE optimizer is based on the Cascades framework [9] but does not exploit common subexpressions. Direct extension of mechanisms proposed earlier for similar subexpressions [10], [11], [12] will generally fail to find the optimal plans. Let us illustrate the problems of applying these techniques to optimize the following SCOPE script.

```
R0 = EXTRACT A,B,C,D FROM "...\test.log"
     USING LogExtractor;
R = SELECT A,B,C,Sum(D) as S FROM R0
    GROUP BY A,B,C;
R1 = SELECT A,B,Sum(S) as S1 FROM R GROUP BY A,B;
R2 = SELECT B,C,Sum(S) as S2 FROM R GROUP BY B,C;
OUTPUT R1 TO "result1.out";
OUTPUT R2 TO "result2.out";
```

The operator DAG of this script is shown in Figure 1(a). The script initially extracts columns *A* to *D* from file *test.log*. The result of this step is named *R0*. Next, the rows of *R0* are grouped using *A*, *B* and *C* as the grouping attributes. The result of this step is referenced as *R*. The rows of *R* are further aggregated to form *R1* (grouping on *A* and *B*) and *R2* (grouping on *B* and *C*). Finally, The contents of *R1* and *R2* are stored in files *result1.out* and *result2.out*, respectively. If a script has several terminal operators, e.g., the $Output$ operators in the previous script, they are connected by a $Sequence$ operator.



(a) Operator DAG.

(b) Different ways to repartition the output of node 1.

Fig. 1. Simple SCOPE script – operator DAG and repartitioning options.

The SCOPE optimizer [7], [8] constructs plans recursively and bottom up, that is, the plan for a node is built after building the plans for its input nodes. Every time a node is processed, it receives a set of physical requirements that the rows produced by the plan must satisfy and it also generates the physical requirements to be imposed on its input nodes. At each node N the optimizer selects the lowest-cost plan for the expression rooted at N. For example, the grouping operator in node *2* requires its input data to be partitioned on $\{A,B,C\}$ or any subset thereof. Note that if the data is partitioned on $\{B\}$, or any subset of $\{A,B,C\}$, it is also partitioned on $\{A,B,C\}$ [8]. In fact, the SCOPE optimizer specifies partitioning requirements using a range, e.g., $[\emptyset\ ,\ \{A,B,C\}]$. Figure 1(b) shows two ways to repartition at node *2*. Furthermore, without loss of generality, let us assume that the cost of repartitioning on $\{A,B,C\}$ at node *2* is smaller than the cost of repartitioning in any other way.

Node *2* in Figure 1(a) has two consumers, nodes *3* and *4*. The crux of the problem is that the consumers may choose different optimal plans for the subexpression rooted at node *2*. For example, for node *3* the best plan may be to partition on $\{A,B\}$ and for node *4* it may be to partition on $\{B,C\}$. These two partition requirements are incompatible forcing the subexpression to be evaluated twice. Allowing each consumer to make the best local choice does not guarantee an overall optimal plan. What is needed is a way to reconcile competing requirements and guarantee an overall optimal plan, which is what we provide in this paper. We describe our solution in the context of the SCOPE optimizer but the technique is applicable to other types of optimizers as well.

Applying previous work on optimization using similar subexpressions [10], [11], [12] to SCOPE queries will not consistently generate the best global plan. These techniques can *identify* common subexpressions but will select the plan that locally minimizes the cost of the shared subexpression, i.e., the one with repartitioning on $\{A,B,C\}$. Note that in this case, the results of node *2* will need to be repartitioned to execute the grouping operations in nodes *3* and *4*.

The algorithms presented in this paper identify common subexpressions and create additional alternatives that use the same physical properties at the shared node. They consider multiple properties for the shared node and generate, for instance, a plan with repartitioning on $\{B\}$ at node *2*. This plan is not locally optimal but has the potential to be part of the globally optimal plan because the grouping operators on $\{A,B\}$ and $\{B,C\}$ can be executed without additional repartitioning. Note that, since the data is partitioned on $\{B\}$, it is also partitioned on $\{A,B\}$ and $\{B,C\}$ [8].

The remaining part of this paper is organized as follows. Section II presents the related work. Section III presents an overview of the system architecture. Section IV describes the use of query fingerprints to identify common subexpressions. The way physical properties are recorded is explained in section V. Section VI describes the propagation of the information about share nodes and the identification of least common ancestor nodes. Section VII describes the re-optimization phase enforcing physical properties. Section VIII presents several techniques to handle large scripts. Section IX presents experimental results of the use of our framework, and Section X presents the conclusions.

II. Related Work

The presence of similar subexpressions has been exploited in several areas of query processing and optimization. In particular, it has been applied to optimization of multiple related queries [11], [13], [14], [15], [16], [12], and materialized view selection [17], [18].

The problem of optimizing multiple related queries (multi-query optimization) has received significant amount of work [11], [13], [14], [15], [16], [12]. The initial work on multi-query optimization [13], [15], [16] proposed solutions that were not fully integrated with the query optimizer and utilized primarily exhaustive algorithms. The work in [11], [12] identifies common subexpressions as a post-optimization task. Only the best plans of the individual queries are considered, thus missing important optimization opportunities. The work in [14] proposes an approach to multi-query optimization that is integrated into an optimizer built on the Volcano [19] framework. The integration requires many fundamental modifications to the optimizer. Furthermore, the work in [14] uses a greedy approach that fails to consider important optimization cases and does not always find the optimal solution [10]. The work in [17], [18] exploits similar subexpressions to improve the performance of materialized views maintenance. The work in [18] proposes the use of a covering subexpression that covers all similar subexpressions and forces all consumers to use this subexpression As explained in [10], this is not always the best solution. It may, for example, be better to use one covering subexpression for some of the consumers and a different one for other consumers. In our work all possible evaluation plans are compared in a fully cost-based manner. The work in [10] proposed a comprehensive framework to use common subexpressions for multi-query optimization and materialized view selection in conventional databases.

None of the earlier techniques take into account competing physical requirements from different consumers. In fact, they

completely ignore the physical properties of the result and simply consider the result to be a set of rows. The importance of exploiting physical properties such as sort order and partitioning of result sets is well known in traditional optimization. Our main contribution is to show how to trade off competing physical requirements in a way that leads to a globally optimal plan.

The work in [20] presents a parallel algorithm to answer composite aggregate queries. However, this work focuses on the specialized problem of answering correlated aggregations over sliding windows. We focus on the optimization of general queries that (i) may or may not have aggregations, and (ii) are evaluated over the entire content of large input relations.

We present the implementation of our solution in the query optimizer of SCOPE [7], [8]. SCOPE scripts are used extensively at Microsoft for massive data analysis and data mining tasks. The work in [7] presents the main language elements of SCOPE. The work in [8] describes the incorporation of partitioning and parallel plans into the SCOPE optimizer.

III. System Architecture

The framework we present is integrated into the transformation-based optimizer used by SCOPE. Conceptually, the optimizer generates all possible rewritings of a query and selects the one with the lowest estimated cost. The optimization process may generate a large number of query rewritings which are stored in very compact form in a *memo* structure [9]. Each node of the memo is known as a *group* and is composed of a set of equivalent *group expressions*. All the group expressions of a group generate the same set of tuples. Each expression is associated with a single operator. An operator references its children or input expressions using group numbers, i.e., unique group identifiers. Each group in the memo can be referenced by multiple group expressions. The optimization process is performed in multiple phases. Earlier phases use a smaller number of transformation rules than later phases. The optimizer is also given a budget that controls how much time can be spent during optimization. Ideally, the earlier optimization phases should apply fast but high-yielding transformation rules.

Figure 2 illustrates the steps in the processing of a query. The SCOPE script is first parsed by the SCOPE compiler which produces an initial logical operator DAG for the script. The SCOPE optimizer identifies the best physical plan it can find. This best plan is later executed by the lower level Dryad and Cosmos subsystems. Figure 2 also shows the four main steps of the optimization process for queries with common subexpressions. The optimization process consists of two phases. The original SCOPE optimizer uses only one optimization phase (phase 1). Phase 2 is a new phase added to re-optimize the query exploiting the presence of common subexpressions.

Step 1: Identifying common subexpressions. This step is performed before the first optimization phase. In this step, subexpression *fingerprints* are employed to quickly identify common subexpressions. A subexpression fingerprint is a highly compressed representation of a subexpression. Fingerprints for every subexpression are constructed in a bottom-up fashion and stored in a hash table. Then, the hash table is used to identify the common subexpressions. The root group of each identified common subexpression is marked as such.

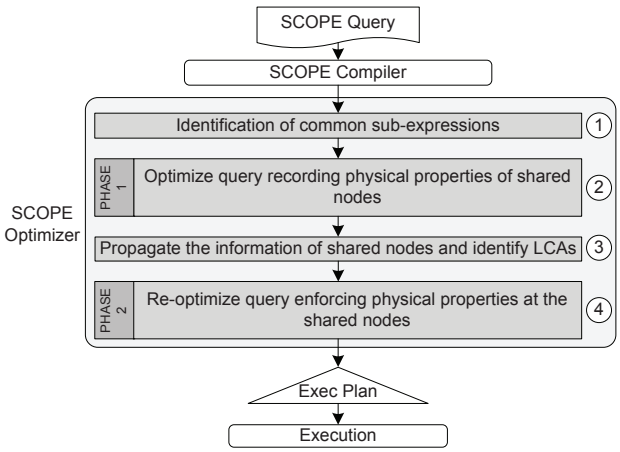


Fig. 2. Query processing steps.

Step 2: Recording physical properties. This step is performed during the original optimization phase, i.e., phase 1. The conventional optimization is extended to record the history of physical properties used in the shared groups identified in Step 1. The history of physical properties is stored as a linked list at every group that is the root of a shared subexpression.

Step 3: Propagating information about shared groups and identifying LCAs. This step is performed right before the re-optimizations begin in phase 2. The information about shared groups is propagated bottom-up from the shared groups to the root. The process also identifies, for each shared subexpression S, the *least common ancestor* group (LCA) of the consumers of S. The LCA of a set of groups or nodes L is the lowest group in the DAG that is traversed by every path from the root to any group in L. The information about shared groups and LCAs is used to guide the final optimization step.

Step 4: Re-optimizing the query enforcing physical properties. This is the new phase added to re-optimize the script exploiting common subexpressions. This step re-optimizes the query enforcing physical properties at the shared groups. When the optimizer processes a group G that is not an LCA, the optimization process continues as usual. When a group G that is the LCA of the consumers of a shared expression S is found, the process re-optimizes the subexpression rooted in G propagating a set of physical properties to be used in S anytime S is optimized. This step considers sub-optimal local plans that can generate a global optimal plan.

IV. Expression Fingerprints

The first step quickly identifies the common query subexpressions and is executed before the first optimization phase. This task computes the fingerprints of all the subexpressions in the script. A fingerprint is a highly accurate summary of an expressions that can be quickly and incrementally

Algorithm 1 IdentifyCommonSubexpressions(M)

Require: Memo M

Ensure: The root groups of all common subexpressions in M are marked as being shared

1. IdentifyExplicitCommSubexpr(M)
2. $H \leftarrow$ empty hash table
3. Compute the fingerprint of each subexpression S in M using a bottom-up traversal of M's groups and store a reference to the root group of S in H using F_S as the hash function
4. **for** every non-empty bucket B of H **do**
5. Compare all the colliding entries of B to identify common subexpressions
6. **if** a set of common subexpressions is identified **then**
7. Remove from M all but one of the subexpressions (R)
8. Add a $SPOOL$ node on top of R and make all the consumers point to this new node
9. Mark the $SPOOL$ node as shared
10. **end if**
11. **end for**

computed using a bottom-up approach. Two expressions that have the same fingerprint are equal with high probability. Two expressions that have different fingerprints are not equal.

Definition 1: (Expression Fingerprints) The *fingerprint* of an expression E rooted in R is denoted as F_E and is computed as follows:

(1) If R represents an operation that directly reads from a data file, then

$$F_E = R.FileID \bmod N$$

(2) Otherwise,

$$F_E = (R.OpID \oplus (\bigoplus_{i=1}^{k} F_{R.child[i]})) \bmod N$$

In Definition 1, $\oplus$ denotes the *XOR* operator. N is a prime number large enough to prevent collisions among the values of $FileID$s and $OpID$s. $FileID$ is the unique identifier of a data file. $OpID$ is the unique identifier of an operation, e.g., all group-by operations have the same $OpID$. If the same operation or file is used in multiple parts of an expression, the same identifier is used in all the occurrences.

Algorithm 1 summarizes the process of identifying all common subexpressions in a query memo. A memo subexpression is a rooted sub-DAG of the memo DAG. Given that all the group expressions of a given group generate the same set of results, the property of being or not shared is assigned to groups. The algorithm identifies in line 1 the common subexpressions explicitly given in the query. A common subexpression is explicitly given when a group is directly referenced from 2 or more different groups, e.g., node 2 in Figure 1(a). Routine $IdentifyExplicitCommSubexpr$ counts the number of different parent groups of each group G of the memo. When the number of parent groups is greater than 1, a $SPOOL$ operator (special SCOPE operator to materialize an intermediate result) is added on top of G, and the $SPOOL$ group is marked as shared. In general, common subexpressions are not always explicitly given. The remaining part of the algorithm identifies all other common subexpressions. This part has two main steps. The first one consists on computing all the memo subexpression fingerprints using a bottom-up approach (lines 2-3). To compute the fingerprint of a memo subexpression S rooted at group G, we use the initial and only group expression that G has at this stage. The properties of the operation associated with the initial group expression of G, e.g., $OpID$ or $FileID$, and the fingerprints of the children groups of this group expression are used to compute the fingerprint of S following Definition 1. The fingerprint of each memo subexpression S is used as the index in a hash table where a reference to the root group of S is stored. Colliding entries in this hash table represent memo subexpressions that are potentially equal. The second step compares, for each non-empty bucket of the hash table, the memo subexpressions corresponding to the colliding entries (lines 4-11). If a set of equal subexpressions is identified, the algorithm removes from the memo all but one of them, adds a $SPOOL$ operator on top of the remaining subexpression, makes all the consumers point to this new group, and marks the $SPOOL$ group as shared.

V. Recording Physical Properties

The next step in the optimization process is the identification of the physical properties that are requested by different consumers at the shared groups. This task is performed during the conventional optimization phase, i.e., phase 1. Every time the optimization routine is called for a shared group during this phase, the routine stores the requested properties for this group. The history of requested properties of a given group G is stored using a linked list. The head of this list is referenced from G. Each node of the list contains a set of required physical properties, that is, partitioning, sorting, and columns.

The transformation-based optimization of an expression can be considered as composed of two steps: logical exploration and physical optimization. The logical exploration step applies the transformation rules of the current optimization phase to generate logically equivalent expressions. The physical optimization step transforms the logical operators, e.g., join and aggregation, to physical operators associated with specific implementation algorithms, e.g., merge join, hash join, hash-based aggregation, and sort-based aggregation.

Algorithm 2 presents a simplified version of the group optimization routine. The algorithm receives a group G, a set of required properties P, and the current optimization phase ph as parameters. The main optimization routine calls *OptimizeGroup* for the root memo group once for each optimization phase. At each phase, calling *OptimizeGroup* for the root group triggers the optimization of the entire query since the routine recursively optimizes the lower level groups.

The operations to record the physical properties of shared groups are presented in lines 1-3. The remaining part of the algorithm corresponds to conventional group optimization. Additional changes to this routine will be introduced in the next sections. The conventional group optimization process generates first the equivalent logical expressions (line 6). Next, implementation rules are used to transform the root nodes of the logical expressions into physical operators (line 8). Then, the child nodes of each physical operator $physE$ are recursively optimized (line 14). The properties to be requested at the child nodes are generated calling the *DetChildProp* (line 12) routine. These properties depend on the type of the physical operator $physE$ and P. The physical plans generated for the child expressions are used to derive the delivered properties of the plan rooted at $physE$ by the *UpdateDlvdProp* routine (line 16). Next, for each physical operator, the routine verifies that the delivered properties satisfies the required properties P calling the *PropertySatisfied* routine (line 18). In general, compensating predicates could be added at this stage to try to generate a plan that satisfy the required properties P. If the required properties are satisfied, the plan is added to the list of valid plans (line 19). The valid plan with the lowest cost is selected (line 23) and compared with the previous best plan of group G (line 24). The best plan of G is updated if needed and returned by the *OptimizeGroup* routine.

Section I stated that partitioning requirements in SCOPE are specified using a range, e.g., [$\emptyset$, {*A*,*B*,*C*}]. It was also observed that a partitioning scheme that generates a local optimal plan may generate a global plan with a higher cost than selecting an alternative partitioning scheme. To help address this problem, every time a partitioning requirement needs to be stored (line 2), our system stores multiple entries corresponding to the different partitioning schemes that satisfy the original requirement. For instance, if an optimization task is called for a shared group with the partitioning requirement [$\emptyset$, {*A*,*B*,*C*}], the system stores the following entries in the history of properties: [{*A*}, {*A*}], [{*B*}, {*B*}], [{*C*}, {*C*}], [{*A*,*B*}, {*A*,*B*}], [{*B*,*C*}, {*B*,*C*}], [{*A*,*C*}, {*A*,*C*}], [{*A*,*B*,*C*}, {*A*,*B*,*C*}]. The way this information is used to find the global optimal plan is presented in Section VII.

Algorithm 2 OptimizeGroup(G, P, ph)

Require: `Group` G, `ReqProp` (required properties) P, `OptPhase` ph

Ensure: The local best plan for G found up to this phase (ph) is returned. In phase 1, the history of properties for shared groups is saved.

1. **if** ph=1 $\wedge$ G is shared $\wedge$ P is not in the history of properties of G **then**
2. Add P to the history of properties of G
3. **end if**
4. `PlanList` $validPlans$
5. /*Logical exploration (applies log. transformations)*/
6. **for** each possible logical expression $logE$ for G using the transformation rules of phase ph **do**
7. /*Physical optimization*/
8. **for** each possible physical implementation $physE$ of the root of $logE$ **do**
9. `DlvdProp` $dlvdProp$
10. **for** each child group C of $physE$ **do**
11. /*Determine required properties for child*/
12. `ReqProp` $cProp \leftarrow$ DetChildProp($physE$, C, P)
13. /*Optimize child expression*/
14. `QueryPlan` $cPlan$ OptimizeGroup(C, $cProp$, ph)
15. /*Update delivered properties of $physE$*/
16. UpdateDlvdProp($dlvdProp$, $cPlan$)
17. **end for**
18. **if** PropertySatisfied(P, $dlvdProp$) **then**
19. Add plan rooted at $physE$ to $validPlans$
20. **end if**
21. **end for**
22. **end for**
23. `QueryPlan` $plan \leftarrow$ cheapestPlan($validPlans$)
24. UpdateBestPlan(G, P, $plan$)
25. return G.bestPlan

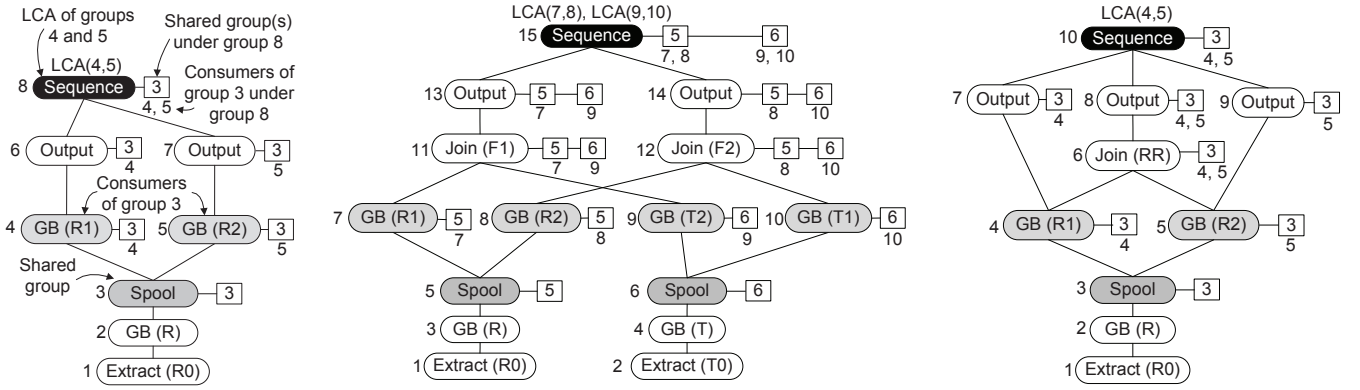
VI. Propagating Information about Shared Groups and Identifying LCAs

The propagation of information about shared groups and the identification of the least common ancestors are performed before the re-optimization phase of the optimization process exploiting common subexpressions. The information about the presence of shared groups is propagated bottom-up from the shared groups to the root group. After the propagation, every group is "aware" of the shared groups below itself. This information will be used to guide the optimization process during the re-optimization phase, i.e., phase 2.

As explained and exemplified in Section I, conventional optimization of a shared group G considers the physical requirements that locally optimize G but does not consider the effect of this selection on the groups that belong to the paths from the consumers of G to the root, i.e., the consuming paths. Furthermore, the different consumers may select different best plans for the shared expression. This would result on an overall query plan that executes the shared subexpressions multiple times.

To address both problems, we re-optimize the expression that contains G and the consuming paths of G while propagating downwards a set of requirements for G that will be enforced every time G is visited. A possible group to start this re-optimization step is the root group since all the consuming paths converge at this group. However, in general, this is not necessary nor efficient. The re-optimization process can start at the lowest group where the consuming paths of the shared group intersect. This special group is known as the least common ancestor (LCA) of the consumers of the shared group.

Definition 2: (Least Common Ancestor) The *Least Common Ancestor (LCA)* of a set of groups or nodes L in a rooted



(a) Script 1: Single shared group

(b) Script 2: Two shared groups

(c) Script 3: Single shared group. LCA is not the lowest common ancestor

Fig. 3. Propagation of information about shared groups and identification of LCAs

operator DAG D is the lowest group in D that is included in every path from an element of L to the root of D.

Figure 3 shows the operator DAGs of three scripts with common subexpressions and the corresponding LCAs. Note that the LCA of a set of groups is not necessarily their lowest common ancestor. Figure 3(c) illustrates this scenario. Group 6 is the lowest common ancestor of groups 4 and 5 but the LCA of these groups is group 10. The reason is that there are paths from the consumer groups to the root that do not contain the lowest common ancestor, e.g., 4-7-10.

Algorithm 3 performs both propagation of the information about shared groups and identification of the LCA of the consumers of each shared group. This algorithm receives a group G as a parameter and recursively processes the groups below G. A group B is said to be below another group A if A belongs to an always ascending path from B to the root. The information about the shared groups below a group G_1 is stored using a list of nodes of type `ShrdGrp`. A list is attached to every group G_1 that has one or more shared groups below itself. Each `ShrdGrp` node of G_1 contains a reference to the associated shared group S and the list of consumers of S that are located below G_1.

The algorithm initially ensures that a group is not processed twice (lines 1-5). If G is a shared group the algorithm adds a `ShrdGrp` node to G (lines 6-10). The associated shared group in this case is G itself. Then, for each input group $inputG$ of G, the algorithm makes a recursive call with $inputG$ as parameter (line 12) and propagates the information of shared groups from $inputG$ to G (lines 14-37). The propagation is performed in the following way. For each shared group $shrdGrpI$ of $inputG$, the algorithm searches a corresponding `ShrdGrp` node $shrdGrpG$ in G (lines 15-27). If a match is found, the algorithm propagates the information about consumer groups from $shrdGrpI$ to $shrdGrpG$ (line 19). After this propagation, the algorithm checks if all the consumers of the shared expression associated with $shrdGrpG$ were found (line 20). If this is the case the algorithm sets G as the LCA of this shared expression (line 22). If another LCA was found previously, the routine *SetLCA* will overwrite that information and set G as the new LCA. If no match for $shrdGrpI$ is found in G, the algorithm adds a copy of $shrdGrpI$ to the list of shared groups of G (line 30). If $inputG$ is a shared group and is also the group associated with $shrdGrpI$, the algorithm records that a consumer of this shared group was found (line 33). The consumer in this case is G itself.

The correctness of Algorithm 3 relies on the following invariant: For each shared group S below a group G (not necessarily the root), after calling the algorithm passing G as parameter, the algorithm has identified the LCA of the consumers of S in the rooted sub-DAG that contains G (root of the sub-DAG) and all the groups below it.

The main optimization routine calls Algorithm 3 passing the root group as the parameter. This single call will propagate the information about all shared groups and will identify the LCA for each shared group. The query expressions in Figure 3 show the content of the lists of `ShrdGrp` nodes attached to each group by Algorithm 3. The graphical notation is presented in Figure 3(a). Figures 3(a) and 3(c) have a single shared group: group 3, while Figure 3(b) has two shared groups: groups 5 and 6. In the latter case, group 15 is the LCA of the consumers of both shared groups.

VII. Re-optimizing While Enforcing Physical Properties

This step re-optimizes the query enforcing physical properties at the shared groups. This task is performed in a new optimization phase, i.e., phase 2, and uses the information generated in previous steps. When the optimization process reaches a group G that is the LCA associated with a shared group S, the process re-optimizes the sub-DAG rooted at G for every possible physical property set that can be enforced at S. This task uses the history of physical properties of S identified in Step 2 (Section V). The optimizer propagates downwards the properties to be enforced at the shared groups. This propagation is done only to the paths that will eventually reach the shared groups and uses the information of shared groups under each group identified in Step 3 (Section VI).

Algorithm 3 PropagateSharedGrpInfoAndFindLCA(G)

Require: `Group` G
Ensure: Information of the shared groups is propagated (bottom-up). The LCA of each shared group is identified.

```
1. if G.alreadyVisited then
2.    return
3. else
4.    G.alreadyVisited ← true
5. end if
6. if G.isShared then
7.    /*add new shared group to G*/
8.    ShrdGrp ng ← newShrdGrp(G.grpNo, G.parents)
9.    G.sharedGroups.add(ng)
10. end if
11. for each input (child) group inputG of G do
12.    PropagateSharedGrpInfoAndFindLCA(inputG)
13.    /*Update info of shared groups of G*/
14.    for each shared group shrdGrpI of inputG do
15.       sharedGrpFound ← false
16.       for each shared group shrdGrpG of G do
17.          if shrdGrpI.grpNo = shrdGrpG.grpNo then
18.             /*propagate information of consumer groups*/
19.             propConsGrps(shrdGrpG, shrdGrpI)
20.             if shrdGrpG.allConsumGrpsFound() then
21.                /*G is a potential LCA of shrdGrpG*/
22.                SetLCA(G, shrdGrpG)
23.             end if
24.             sharedGrpFound ← true
25.             break
26.          end if
27.       end for
28.       if !sharedGrpFound then
29.          /*add shrdGrpI to list of shared groups of G*/
30.          ShrdGrp ngI ← copyShrdGrp(shrdGrpI)
31.          if inputG.grpNo = shrdGrpI.grpNo then
32.             /*G is a consumer of inputG*/
33.             ngI.setConsumGrpAsFound(G);
34.          end if
35.          G.sharedGroups.add(ngI)
36.       end if
37.    end for
38. end for
```

Since the property set to be enforced in S is propagated from the LCA G, all paths that reach S enforce the same property in S. This ensures that the resulting plan will execute the common expression once. Furthermore, the plans obtained at the LCA G consider the effects of the physical properties of S on the consuming paths. This allows finding the global optimal plan for G. The changes to implement this step are presented in algorithms 4 and 5. These algorithms are an updated version of the routine *OptimizeGroup* presented previously in Algorithm 2. The code of the original routine has been divided in two routines to simplify the presentation.

Algorithm 4 presents the extended *OptimizeGroup* routine.

Algorithm 4 OptimizeGroup(G, P, ph)

Require: `Group` G, `ExtReqProp` (extended required properties, includes properties to be enforced at the shared groups) P, `OptPhase` ph
Ensure: The best plan for G found up to this phase (ph) is returned. In phase 1, the history of properties for shared groups is saved. Phase 2 optimizes exploiting common sub-expressions.

```
1. if ph=1 ∧ G is shared ∧ P is not in the history of properties of G then
2.    Add P to the history of properties of G
3. end if
4. if ph=2 ∧ G is the LCA corresponding to shared groups ShrdG then
5.    /*Optimization enforcing physical properties*/
6.    PlanList newValidPlans
7.    for each combination ShrdGrpProps of physical properties for ShrdG do
8.       SetPropForSharedGrps(P, ShrdGrpProps)
9.       QueryPlan newPlan ← LogPhysOpt(G, P, ph)
10.      Add newPlan to newValidPlans
11.   end for
12.   QueryPlan plan ← cheapestPlan(newValidPlans)
13. else
14.   QueryPlan plan ← LogPhysOpt(G, P, ph)
15. end if
16. UpdateBestPlan(G, P, plan)
17. return G.bestPlan
```

This routine receives three parameters: G, the group to optimize, P, the set of required properties, and ph, the current optimization phase. The only change on the parameters is on the data type of the second parameter (P), which is now of type `ExtReqProp`. `ExtReqProp` is an extension of `ReqProp` that has an additional field named *PropForSharedGrps*. This field is used to store the properties to be enforced at the shared groups. Lines 1-3 of Algorithm 4 record the history of properties at the shared groups during phase 1. If the group being optimized, i.e., G, is the LCA associated with one or more shared groups and the optimization phase is 2, then the routine re-optimizes G enforcing physical properties at the shared groups (lines 6-12). Otherwise, it calls the routine *LogPhysOpt* to perform regular logical and physical optimization of G (line 14). To re-optimize enforcing physical properties, the *OptimizeGroup* routine generates all possible combinations of the physical property sets to be enforced at the shared groups (line 7). The combinations of property sets are generated changing initially the property sets for the first shared group and using initial property sets for the remaining groups. Then, the next property set is assigned to the second group and a new combination is generated for every possible property set that can be assigned to the first group, and so on. For instance, if G is the LCA of groups 3 and 4 and their history of property sets are $\{p1,\ p2\}$ and $\{q1,\ q2\}$,

respectively, the routine generates the following combinations of properties: {(3, $p1$), (4, $q1$)}, {(3, $p2$), (4, $q1$)}, {(3, $p1$), (4, $q2$)} and {(3, $p2$), (4, $q2$)}. The generation of combinations of properties is improved in Section VIII to handle large scripts. For each possible combination, the *OptimizeGroup* routine sets the properties to be enforced in the *PropForSharedGrps* field of P (line 8) and re-optimizes G calling *LogPhysOpt* (line 9). *LogPhysOpt* will enforce the physical properties at the shared groups. We refer to each re-optimization process as a round. The new plans obtained from the different rounds are added to a list of valid plans (line 10) and, after processing all the rounds, the routine selects the plan with the smallest cost (line 12). The plan found in lines 12 or 14 is compared with the previous best plan of G. The best plan of G is updated if needed (line 16) and returned (line 17).

As stated in Section III, the actual implementation of the optimization routines uses an optimization budget that controls the time to be spent during optimization. We extended this feature to the new optimization phase to allow the optimizer to stop at an intermediate round and use the best plan found up to that point.

Algorithm 5 presents the *LogPhysOpt* routine. This routine has the same parameters as Algorithm 4 and is an extended version of the code in lines 4-24 of the original *Optimize-Group* routine (Algorithm 2). The routine performs the logical exploration and physical optimization of a given group G and returns the plan for G with the lowest cost. In addition, in phase 2, the routine also propagates downwards the properties to be enforced at the shared groups and enforces these properties when it reaches the shared groups. The difference with respect to the code of the original *OptimizeGroup* is on the way the routine determines the required properties for the child groups of G (lines 9-17). Line 9 initializes the properties $cProp$ that will be used to optimize the child being considered (C). If the current phase is 2 and C is shared, the routine enforces the physical property set propagated from the LCA for C (line 11). This operation will load the conventional fields of $cProp$ with the information of the property set to be enforced at C. This information is obtained from the *PropForSharedGrps* field of P. If the current phase is not 2 or C is not shared, the conventional fields of $cProp$ are loaded as in the original *OptimizeGroup* routine, i.e., calling the routine *DetChildProp* (line 13). In phase 2, the routine also propagates downwards the properties for shared groups from P to $cProp$ (line 16). The routine *PropagPropForSharedGrps* propagates all the property sets from the field *PropForSharedGrps* of P to the one of $cProp$. The only property set that is not propagated is the one for the group child C when C is shared. In this case, a shared group has been reached.

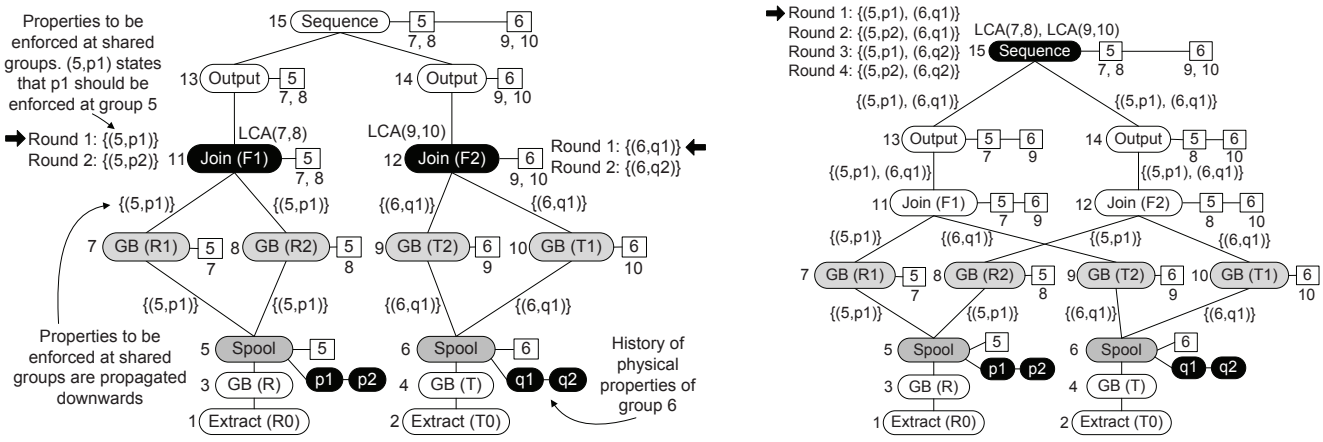
Figure 4 shows the operator DAGs of two SCOPE scripts. The operator DAG in Figure 4(a) has two shared groups that are associated with different LCAs. The one in Figure 4(b) has two shared groups associated with the same LCA. These figures represent the operation of the re-optimization phase based on Algorithms 4 and 5. The figures show the information about shared groups under each group. This is presented using the same graphical notation explained in Figure 3(a). In addition, they show the properties to be enforced at the shared groups for each possible round, the history of properties of the shared groups, and the propagation of the properties to be enforced at the shared groups during round 1. Figure 4(a) presents the notation of the additional graphical elements. The label attached to an edge that is not incident to a shared group represents the property sets for shared groups being propagated. If an edge is incident to a shared group, the label represents the property set being enforced at the shared group. For example, in the DAG of Figure 4(a), group 11 is the LCA associated with shared group 5. The history of physical properties of group 5 has two property sets $p1$ and $p2$. When group 11 is reached in phase 2, the optimizer generates

Algorithm 5 LogPhysOpt(G, P, ph)

Require: `Group` G, `ExtReqProp` (extended required properties, includes properties to be enforced at the shared groups) P, `OptPhase` ph

Ensure: Performs logical exploration and physical optimization. In phase 2, propagates downwards the properties to be enforced at the shared groups

```
1.  PlanList validPlans
2.  /*Logical exploration (applies log. transformations)*/
3.  for each possible logical expression logE for G using the
    transformation rules of phase ph do
4.    /*Physical optimization*/
5.    for each possible physical implementation physE of
      the root of logE do
6.      DlvdProp dlvdProp
7.      for each child group C of physE do
8.        /*Determine required properties for child*/
9.        ExtReqProp cProp
10.       if ph=2 ∧ C is shared then
11.         cProp ← EnforcePhysProp(C, P)
12.       else
13.         cProp ← DetChildProp(physE, C, P)
14.       end if
15.       if ph=2 then
16.         PropagPropForSharedGrps(cProp, C, P)
17.       end if
18.       /*Optimize child expression*/
19.       QueryPlan cPlan OptimizeGroup(C, cProp,
          ph)
20.       /*Update delivered properties of physE*/
21.       UpdateDlvdProp(dlvdProp, cPlan)
22.     end for
23.     if PropertySatisfied(P, dlvdProp) then
24.       Add plan rooted at physE to validPlans
25.     end if
26.   end for
27. end for
28. QueryPlan plan ← cheapestPlan(validPlans)
29. return plan
```



(a) Script 1: Two shared groups with different LCAs

(b) Script 2: Single LCA for two shared groups

Fig. 4. Re-optimization enforcing physical properties

two optimization rounds corresponding to the two possible combinations of properties for group 5: $\{(5,\ p1)\}$ and $\{(5,\ p2)\}$. The label $\{(5,\ p1)\}$ attached to the edge between groups 11 and 7 represents that this requirement is propagated from group 11 to 7 during round 1. The label $\{(5,\ p1)\}$ attached to the edge between groups 7 and 5 represents that property set $p1$ is being enforced to optimize group 5 during round 1.

The plans generated in the re-optimization phase (phase 2) are different from the ones generated in the initial optimization pase (phase 1). The optimizer will select the plan with the lowest cost. This plan could have been generated in any phase.

VIII. Handling Large Scripts

If the optimization budget is sufficiently large, the optimization process described in the previous sections evaluates all the possible plans and finds the global optimal plan. In real world scripts, when the number of groups and shared groups is not very large, the process will potentially find a good global solution that executes the shared subexpressions once using a similar total budget as the one used for normal query optimization. However, scripts may also have a large number of groups and shared groups. Under this scenario, the presented optimization routines would generate a very large number of optimization rounds during the re-optimization phase. In this section we present several extensions of the proposed framework to handle large scripts. The proposed extensions aim to reduce the number of rounds and to evaluate the most promising ones early.

A. Exploiting Independent Shared Groups

When several shared groups are associated with the same LCA group L, the number of rounds to optimize L can be dramatically reduced if the shared groups are independent as specified in the following definition.

Definition 3: (Independent Shared Groups) A set of shared groups that are associated with the same LCA group L are $independent$ if the sub-DAGs constructed for each shared group S including all the groups in the consuming paths of S share only the group L and possibly groups above L.

Figure 5 shows an operator DAG with two independent shared groups (5 and 6). The identification of independent shared groups is performed after completing Step 3 (Section VI) and uses the information of shared groups under each input group of LCAs. Note that, if a group L is the LCA associated with two shared groups G_1 and G_2, has two input groups I_1 and I_2, and the lists of shared groups below I_1 and I_2 are $ShrdGrpsI_1$ and $ShrdGrpsI_2$, respectively; G_1 and G_2 are independent if none of the lists ($ShrdGrpsI_1$, $ShrdGrpsI_2$) contains both shared groups (G_1 and G_2). We can generalize this property to identify the disjoint independent sets of shared groups for any LCA L as follows. We create a copy of the sets of shared groups under each input of L and use this copy in the next steps. Next, we remove all the groups that do not have L as their associated LCA. Then, we iteratively merge the sets that share at least one element. The final sets represent the independent sets of shared groups.

If multiple shared groups are independent, they can be re-optimized independently. To do this, the way rounds are generated (line 7 of Algorithm 4) is modified as follows. The optimizer generates first the rounds for the different property sets that can be enforced in the first shared group while using the initial property sets for the other groups. When these rounds are completed, the optimizer has identified the best property set $bestP_1$ to optimize the first group. Further rounds will only use $bestP_1$ as the property set for the first group. The process continues generating rounds for the different property sets of the second shared group and so on.

Let us consider for example the expression with two shared groups (5 and 6) illustrated in Figure 5. Group 5 has eight property sets in its history of properties: $p1$ to $p8$. Group 6 has also eight property sets in its history: $q1$ to $q8$. Without the changes presented in this section, the $OptimizeGroup$ routine generates, at group 15, as many rounds as the number of combinations of properties for the shared groups. This

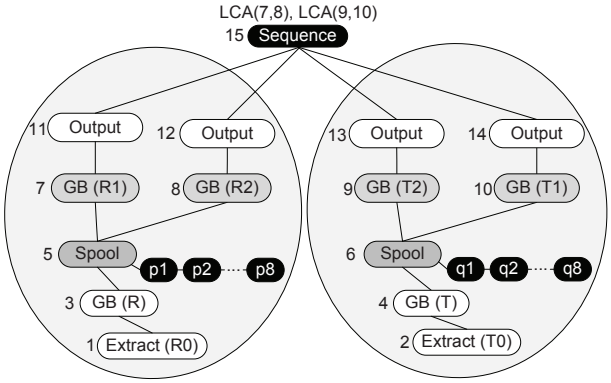


Fig. 5. Two independent shared groups.

would generate 64 rounds. However, since groups 5 and 6 are independent shared groups, the extended routine will generate the following rounds. A round is generated for each of the eight property sets that can be enforced in group 5 while using the initial property set $q1$ for group 6. This step will generate rounds for the following required properties: $\{(5, p1), (6, q1)\}$, $\{(5, p2), (6, q1)\}$, ..., $\{(5, p8), (6, q1)\}$. After these rounds are executed, the routine has identified the best property P_{best} for group 5, i.e., the property that generated the smallest cost in the first eight rounds. Next, the routine generates rounds for the properties that can be enforced in group 6 while using always P_{best} for group 5. This generates rounds for the following required properties: $\{(5, P_{best}), (6, q2)\}$, $\{(5, P_{best}), (6, q3)\}$, ..., $\{(5, P_{best}), (6, q8)\}$. After this new set of rounds, the routine will have identified the best combination of properties to re-optimizes group 15. The extended routine generates only 15 rounds.

B. Ranking Shared Groups

The goal of this extension is to perform the more promising rounds early. This extension considers the fact that the optimization of different shared groups may provide different reductions in the overall plan cost. The potential repartitioning savings of a group is used as an indicator of how beneficial optimizing the groups is.

$$RepartSav(G) = (NoConsumers(G)-1)*RepartCost(G)$$

Note that the more consumers a group has and the higher the repartitioning cost associated with the group is, the higher the repartitioning savings will be. When a given group L is the LCA associated with multiple shared groups, the shared groups are ranked based on their repartitioning savings (high to low). The new order of the shared groups is used to generate the optimization rounds in line 7 of Algorithm 4. Given that the $OptimizeGroup$ routine generates first the rounds for the property sets to be enforced at the first shared groups, the process will identify early the plans that optimize these highly beneficial groups.

C. Ranking Shared Group Properties

Similarly to the technique presented in Section VIII-B, this extension aims to improve the generation of re-optimization rounds to process the more promising ones early. In this case, we use the fact that multiple property sets in the history of properties of a shared group can yield different benefits in the overall plan cost. We use the number of times a property set P generated a best local plan during phase 1 as an indicator of how beneficial P can be in phase 2. For each shared group S, we rank the elements in its history of properties from high to low associated frequency. This task is performed right before the re-optimization phase. The new order of elements in the history of property sets is used during the generation of rounds (line 7 of Algorithm 4). The optimizer will evaluate the rounds for potentially highly beneficial properties first.

IX. Experimental Results

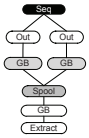
We have extended Microsoft's SCOPE optimizer with the algorithms for exploiting common subexpressions presented in Sections III to VII. We also implemented the techniques to handle large scripts discussed in Section VIII. This section presents several experimental results of our implementation. We run the SCOPE optimizer on an Intel Dual Core 2GHz machine with 2GB RAM using Microsoft Windows as operating system.

Figure 6 shows the scripts used in the experimental evaluation. Scripts $S1$ to $S4$ are small SCOPE scripts designed to test different configurations of shared groups and their associated LCA groups. Figure 6 presents the text and the corresponding operator DAGs of these scripts. $S1$ is the motivating script used in Section I. It has a single shared group with two consumers. $S2$ is similar to $S1$ but it has three consumers. $S3$ has two shared groups with different associated LCAs. $S4$ has also two shared groups but they have the same associated LCA group. The shared groups of $S4$ are non-independent. Scripts $LS1$ and $LS2$ are large real world SCOPE scripts. They are both used to analyze large logs generated by Microsoft online services. Figure 6 presents important properties of these scripts. The initial memo structure of $LS1$ has 101 groups, 4 of them are shared groups. $LS2$ is significantly larger than $LS1$, its initial memo has 1034 groups and 17 shared groups.

Figure 7 presents the results of optimizing the queries of Figure 6 with and without the framework to exploit common subexpressions. The presented estimated costs are computed by the optimizer using techniques to accurately represent query execution times. Note that these cost estimation techniques are not modified in this paper. The cost of the plan for $S1$ using only conventional optimization is 8185. The cost using our framework to exploit common subexpressions is 5037, this is only 62% of the original cost. Figure 8 shows the plans for $S1$ generated by the conventional and extended optimizers. The edges with multiple arrows represent that multiple instances of the operators are running in parallel on multiple machines. The shaded groups represent operations that exchange data among the cluster machines. These operations are in general very costly. A sequence of non-shaded operators where every

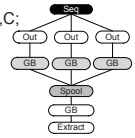
S1: Single shared group with two consumers

```
R0 = EXTRACT A,B,C,D FROM "...\test.log"
       USING LogExtractor;
R = SELECT A,B,C,Sum(D) as S FROM R0 GROUP BY A,B,C;
R1 = SELECT A,B,Sum(S) as S1 FROM R GROUP BY A,B;
R2 = SELECT B,C,Sum(S) as S1 FROM R GROUP BY B,C;
OUTPUT R1 TO "result1.out";
OUTPUT R2 TO "result2.out";
```



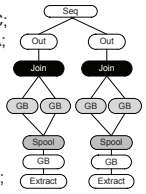
S2: Single shared group with three consumers

```
R0 = EXTRACT A,B,C,D FROM "...\test.log"
       USING LogExtractor;
R = SELECT A,B,C,Sum(D) as S FROM R0 GROUP BY A,B,C;
R1 = SELECT B,A,Sum(S) as S1 FROM R GROUP BY B,A;
R2 = SELECT A,C,Sum(S) as S2 FROM R GROUP BY A,C;
R3 = SELECT A,Sum(S) as S3 FROM R GROUP BY A;
OUTPUT R1 TO "result1.out";
OUTPUT R2 TO "result2.out";
OUTPUT R3 TO "result3.out";
```



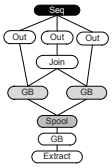
S3: Two shared groups with different LCA

```
R0 = EXTRACT A,B,C,D FROM "...\test.log"
       USING LogExtractor;
R = SELECT A,B,C,Sum(D) as S FROM R0
     GROUP BY A,B,C;
R1 = SELECT B,C,Sum(S) as S1 FROM R GROUP BY B,C;
R2 = SELECT B,A,Sum(S) as S2 FROM R GROUP BY B,A;
RR = SELECT R1.B,A,C,S1,S2 FROM R1,R2
       WHERE R1.B=R2.B;
T0 = EXTRACT A,B,C,D FROM "...\test2.log"
       USING LogExtractor;
T = SELECT A,B,C,Sum(D) as S FROM T0
     GROUP BY A,B,C;
T1 = SELECT B,C,Sum(S) as S1 FROM T GROUP BY B,C;
T2 = SELECT B,A,Sum(S) as S2 FROM T GROUP BY B,A;
TT = SELECT T1.B,A,C,S1,S2 FROM T1,T2
       WHERE T1.B=T2.B;
OUTPUT RR TO "result1.out";
OUTPUT TT TO "result2.out";
```



S4: Two non-independent shared groups

```
R0 = EXTRACT A,B,C,D FROM "...\test.log"
       USING LogExtractor;
R = SELECT A,B,C,Sum(D) as S FROM R0 GROUP BY A,B,C;
R1 = SELECT B,C,Sum(S) as S1 FROM R GROUP BY B,C;
R2 = SELECT B,A,Sum(S) as S2 FROM R GROUP BY B,A;
RR = SELECT R1.B,A,C FROM R1,R2
       WHERE R1.B=R2.B;
OUTPUT R1 TO "result1.out";
OUTPUT R2 TO "result2.out";
OUTPUT RR TO "result3.out";
```



LS1: Large real-world script 1

4 shared groups
3 of them have 2 consumers, 1 has 3 consumers
101 operators in the initial operator DAG (before phase 1)

LS2: Large real-world script 2

17 shared groups
15 of them have 2 consumers, 1 has 4 consumers, 1 has 5 consumers
1034 operators in the initial operator DAG (before phase 1)

Fig. 6. Experimental evaluation scripts.

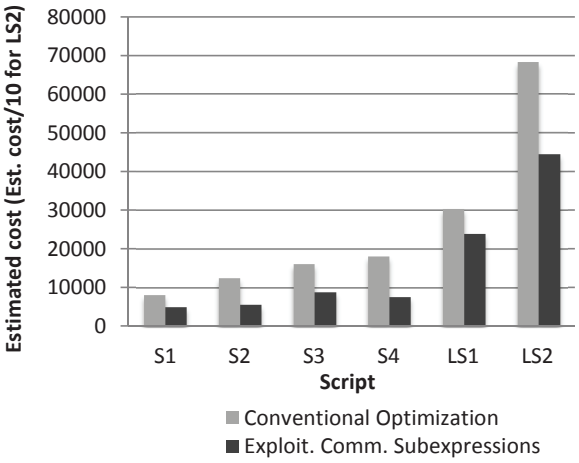
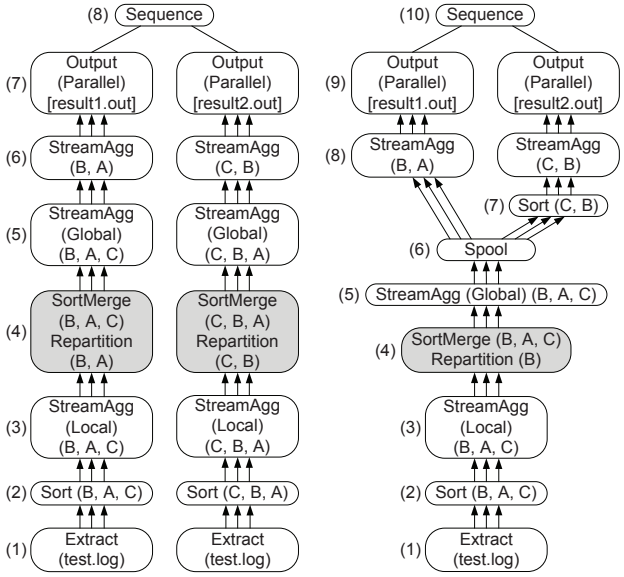


Fig. 7. Experimental evaluation results.



(a) Conventional optimization. (b) Optimization exploiting common subexpressions.

Fig. 8. Script plan comparison for $S1$.

pair is connected with multiple arrows represent that multiple machines execute the sequence of operations. A $Sequence$ operator on top of a plan does not process the data generated by its input operators. It just specifies that the overall plan is composed of several sub-plans.

The conventional optimizer generates a plan that executes the common subexpression twice (Figure 8(a)). Each time the shared group is reached by the $OptimizeGroup$ routine, the requirements to optimize this group are different and consequently the plans generated for the common subexpression are also different. The plan is composed of two independent subplans that perform the same sequence of operations, (1) to (7), but using different attributes.

- (1) to (3). $test.log$ is partitioned and distributed twice across all machines in the cluster. After extracting the attributes A, B, C and D, the input is sorted and aggregated (at each machine) on attributes {B,A,C} in the left subplan and {C,B,A} in the right subplan.
- (4). To group all the data on {A,B,C}, all the records with the same values of A, B and C should be located in the same machine. This is obtained repartitioning on {B,A} in the left subplan and on {C,B} in the right one. Next, each machine sorts its data on {B,A,C} in the left subplan and on {C,B,A} in the right one.
- (5). Each machine aggregates its data grouping on {B,A,C} in the left subplan and on {C,B,A} on the right subplan. This step computes the values of S.
- (6). Each machine further aggregates its intermediate data grouping on {B,A} in the left subplan and on {C,B} on the right subplan. This step computes the values of $S1$ and $S2$ in the left and right subplans, respectively.

- (7). The resulting data is stored in parallel building the distributed files $result1.out$ and $result2.out$.

Our extended optimizer generates a plan that executes the common subexpression once and uses its results twice (Figure 8(b)).

- (1) to (3). The input file is partitioned and distributed once across all machines in the cluster. After extracting the attributes A, B, C and D, the input is sorted and aggregated on {B,A,C} at each machine.
- (4). The data is repartitioned once on {B}. After that, each machine sorts its data on {B,A,C}.
- (5). Each machine aggregates its data grouping on {B,A,C}. The values of S are computed.
- (6). The intermediate results are materialized.
- (7) and (8). The left operator further aggregates the materialized data on {B,A}. The right operators sort and aggregate the materialized data on {C,B}. $S1$ and $S2$ are computed in (8).
- (9). The resulting data is stored in parallel building the distributed files $result1.out$ and $result2.out$.

Figure 7 shows also that the extended optimizer reduces the cost of $S2$ by 55%. This result shows that the reduction depends on the number of consumers. The reductions on the cost of the two-shared-group scripts $S3$ and $S4$ are 45% and 57%, respectively. The execution time of the optimization process for queries $S1$ to $S4$ was smaller than one second. Our tests also showed significant saving on the large real world scripts. The savings on scripts $LS1$ and $LS2$ are of 21% and 45%, respectively. The optimization of these scripts benefited from the extensions to handle large scripts. The time budgets to optimize $LS1$ and $LS2$ were of 30 and 60 seconds, respectively. In all the evaluated scripts the overall time budged used for optimization is a small fractions of the execution time of the produced plan (smaller than 1%).

X. Conclusions

Scripts for cloud-based massive data analysis, e.g., using Microsoft's SCOPE or Yahoo!'s Pig Latin, often contain common subexpressions. The optimization of these scripts using conventional query optimization techniques generates plans that execute the common subexpressions as many times as they are consumed. In this paper, we present a framework to correctly optimize cloud scripts that contain common subexpressions. The framework produces plans that execute common expressions only once. The optimization process is extended with a new re-optimization phase that enforces physical properties at the shared groups. Our approach reconciles competing physical requirements in a way that leads to a globally optimal plan. Our framework was designed to be integrated with optimizers that use the common Cascades model. We prototyped the framework in SCOPE and the experimental analysis shows that it reduced significantly (from 21 to 57%) the estimated cost of simple and large real-world scripts.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03: Proceedings of the 19th ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 29–43.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[3] ——, "Mapreduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[5] Apache, "Hadoop," http://hadoop.apache.org/.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.

[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, 2008.

[8] J. Zhou, P.-A. Larson, and R. Chaiken, "Incorporating partitioning and parallel plans into the scope optimizer," in *ICDE '07: Proceedings of 26th IEEE International Conference on Data Engineering, 2010*, 2010, pp. 1060–1071.

[9] G. Graefe, "The cascades framework for query optimization," *IEEE Data Engineering Bulletin*, vol. 18, no. 3, pp. 19–29, 1995.

[10] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2007, pp. 533–544.

[11] S. Finkelstein, "Common expression analysis in database applications," in *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1982, pp. 235–245.

[12] S. N. Subramanian and S. Venkataraman, "Cost-based optimization of decision support queries using transient-views," in *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1998, pp. 319–330.

[13] J. Park and A. Segev, "Using common subexpressions to optimize multiple queries," in *ICDE '88: Proceedings of the 4th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1988, pp. 311–319.

[14] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2000, pp. 249–260.

[15] T. K. Sellis, "Multiple-query optimization," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 23–52, 1988.

[16] T. Sellis and S. Ghosh, "On the multiple-query optimization problem," *IEEE Trans. on Knowl. and Data Eng.*, vol. 2, no. 2, pp. 262–266, 1990.

[17] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, "Materialized view selection and maintenance using multi-query optimization," in *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2001, pp. 307–318.

[18] W. Lehner, R. Cochrane, H. Pirahesh, and M. Zaharioudakis, "fast refresh using mass query optimization," in *Proceedings of the 17th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 391–398.

[19] G. Graefe and W. J. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *Proceedings of the Ninth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 209–218.

[20] L. Chen, C. Olston, and R. Ramakrishnan, "Parallel evaluation of composite aggregate queries," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 218–227.