In-memory Caching for Multi-query Optimization of Data-intensive Scalable Computing Workloads

Pietro Michiardi
EURECOM
Biot, France
pietro.michiardi@eurecom.fr

Damiano Carra
University of Verona
Verona, Italy
damiano.carra@univr.it

Sara Migliorini
University of Verona
Verona, Italy
sara.migliorini@univrit

ABSTRACT

In modern large-scale distributed systems, analytics jobs submitted by various users often share similar work. Instead of optimizing jobs independently, multi-query optimization techniques can be employed to save a considerable amount of cluster resources.

In this work, we introduce a novel method combining in-memory cache primitives and multi-query optimization, to improve the efficiency of data-intensive, scalable computing frameworks. By careful selection and exploitation of common (sub) expressions, while satisfying memory constraints, our method transforms a batch of queries into a new, more efficient one which avoids unnecessary recomputations. To find feasible and efficient execution plans, our method uses a cost-based optimization formulation akin to the multiple-choice knapsack problem. Experiments on a prototype implementation of our system show significant benefits of worksharing for TPC-DS workloads.

1 INTRODUCTION

Modern technologies to analyze large amounts of data have flourished in the past decade, with general-purpose cluster processing frameworks such as MapReduce [1], Dryad [2] and Spark [3]. More recently, a lot of effort has been put in raising the level of abstraction, and allow users to interact with such systems with a relational API. SQL-like querying capabilities are not only interesting to users for their simplicity, but also bring additional benefits from a wide range of automatic query optimizations, aiming at efficiency and performance.

Large-scale analytics systems are deployed in shared environments, whereby multiple users submit queries concurrently. In this context, concurrent queries often perform similar work, such as scanning and processing the same set of input data. The research in [4] on 25 production clusters, estimated that over 35,000 hours of redundant computation could be eliminated per day by simply reusing intermediate query results (approximately equivalent to shutting off 1500 machines daily). It is thus truly desirable to study query optimization techniques that go beyond optimizing the performance of a single query, but instead consider multiple queries, for efficient resource utilization.

Multi-query optimization (MQO) amounts to find similarities among a set of queries and uses a variety of techniques to avoid redundant work during query execution. For traditional database systems, MQO trades some small optimization overheads for increased query performance, using techniques such as sharing sub-expressions [5–7], materialized views selection [8, 9], and pipelining [10]. Work sharing optimizations operating at query runtime, for staged databases, have also been extensively studied [11–14]. The idea of reusing intermediate data across queries running in distributed systems has received significant attention: for MapReduce [15, 16], for SCOPE operating on top of Cosmos [17] and for Massive Parallel Processing frameworks [18].

In this paper, we study MQO in the context of distributed computing engines such as Apache Spark [3], with analytics jobs written in SparkSQL [19], in which relational operators are mapped to stages of computation and I/O. Following the tradition of RDBMSes, queries are first represented as (optimized) logical plans, which are transformed into (optimized) physical plans, and finally run as execution plans. Additionally, modern parallel processing systems, such as Spark, include an operator to materialize in RAM the content of a (distributed) relation, which we use extensively. Our approach to MQO is that of traditional database systems, as it operates on a batch of queries. However, unlike traditional approaches, it blends pipelining and global query planning with shared operators, using *in-memory* caching to support worksharing. Our problem formulation amounts to a *cache admission problem*, which we cast as a cost-based, constrained combinatorial optimization task, setting it apart from previous works in the literature.

We present the design of a MQO component that, given a set of concurrent queries, proceeds as follows. First, it analyzes query plans to find sharing opportunities, using logical plan fingerprinting and an efficient lookup procedure. Then it builds multiple *sharing plans*, using shared relational operators and scans, which subsume common work across the given query set. Sharing plans materialize their output relation in RAM. A cost-based optimization selects best sharing plans with dynamic programming, using cardinality estimation and a knapsack formulation of the problem, that considers a memory budget given to the MQO problem. The final step is a global query plan rewrite, including sharing plans which pipeline their output to modified consumer queries.

We validate our system with a prototype built for SparkSQL, using the standard TPC-DS benchmark for the experiments. Overall, our method achieves up to 80% reduction in query runtime, when compared to a setup with no worksharing. Our main contributions are as follows:

• We propose a general approach to MQO for distributed computing frameworks. Our approach produces *sharing plans*, that are materialized in RAM, aiming at eliminating redundant work.
• We cast the optimization problem of selecting the best sharing plans as a Multiple-choice Knapsack problem, and solve it efficiently through dynamic programming.
• Our ideas materialize into a system prototype based on SparkSQL, which we evaluated using standard benchmarks, obtaining tangible improvements in aggregate query execution times.

2 RELATED WORK

MQO in RDBMSes. Multi-query optimization has been studied extensively [6, 7, 10, 20, 21]. More recently, similar subexpressions sharing has been revisited by Zhou et al. in [5], who show that reusable common subexpressions can improve query performance. Their approach avoids some limitations of earlier work

[7, 20]. More recently, work sharing at the execution engine has been studied [11–14]. The MQO problem is considered at query runtime, and requires a staged database system. Techniques such as pipelining [12] and multiple query plans [22, 23] have proven extremely beneficial for OLAP workloads. Our work is rooted on such previous literature, albeit the peculiarities of the distributed execution engine (which we also take into account in our cost model), and the availability of an efficient mechanism for distributed caching steer our problem statement apart from the typical optimization objectives and constraints from the literature.

Materialized views can be used in conjunction with MQO to reduce query response times [8, 9, 24]. A broad range of works addressed the problem of materialized view selection and maintenance, including both deterministic [7, 25, 26] and randomized [27–29] strategies. In this paper, we focus on analytics queries in which data can be assumed to be static. Workloads consist mostly of ad-hoc, long running, scan-heavy queries over data periodically loaded in a distributed file system. Problems related to view maintenance do not manifest in our setup. Moreover, our approach considers storing intermediate relations in RAM.

MQO in Cloud and Massively Parallel Processing (MPP). Building upon MQO techniques in RDBMSes, Silva et al. [17] proposed an extension to the SCOPE query optimizer which optimizes cloud scripts containing common expressions while reconciling physical requirements. In MPP databases, the work in [18] presents a comprehensive framework for the optimization of Common Table Expressions (CTEs) implemented for Orca. Compared to our method, we consider not only CTEs but also similar subexpressions to augment sharing opportunities.

MQO in MapReduce. The idea of making MapReduce jobs share some intermediate results was studied in [15–17, 30–34]. The common denominator of such works is that they operate at a lower level of abstraction than we do.

More recently, [35] formulates the MQO problem as a cost-based, binary optimization task, which is addressed with an external optimizer. Nevertheless, this work does not exploit caching as a mechanism to re-use work. The work in [36] presents a method to estimate the benefit associated to materializing intermediate results of past queries, and this method is orthogonal to ours. Additionally, views are materialized on disk, instead of memory. The work in [37] addresses the problem of work sharing by casting it as an *exact* query subgraph matches. As such, it tackles the MQO problem from a different angle, and it does not exploit caching. The work in [38] cast the MQO task as an integer linear programming problem. However, the induced cost model is simplistic, and does not exploit in-memory caching.

Caching to recycle work. Previous works [39–43] that address the problem of reusing intermediate query results, cast it as a general caching problem. Our work substantially differs from those approaches in that they mainly focus on cache *eviction*, where past queries are used to decide what to keep in memory, in an on-line fashion. Instead, in this work we focus on the off-line constrained optimization problem of cache *admission*: the goal is to decide the best content to store in the cache, rather than selecting which to evict if space is needed. The only work that considers the reuse of intermediate results when analyzing the overall execution plan of *multiple* queries is [40]. Nevertheless, they focus on small problem instances which do not require the general, cost-based approach we present in this work.

3 PROBLEM STATEMENT

We introduce a simple running example, that is rich enough to illustrate the gist of the MQO problem. Consider the following three concurrent queries:

```
QUERY 1:
SELECT name, dept_name, salary
FROM employees, departments, salaries
WHERE dep = dept_id
    AND id = emp_id
    AND gender = 'F'
    AND location = 'us'
    AND salary > 20000
ORDER BY salary DESC

QUERY 2:
SELECT name, dept_name, title,
    to as title_expired_on
FROM departments, employees, titles
WHERE dep = dept_id
    AND id = emp_id
    AND gender = 'F'
    AND location = 'us'
    AND from >= 2010

QUERY 3:
SELECT id, name, salary, from_date
FROM employees, salaries
WHERE id = emp_id
    AND age > 30
    AND SALARY > 30000
```

Figure 1 illustrates the optimized operator trees (logical plans) of the queries in the above example. The leaf nodes represent the base relations. Each intermediate node is a relational operator (*Selection, Projection, Join*, etc.). The arrows between nodes indicate data flow. Our MQO strategy uses such optimized logical plans to produce new plans – whose aim is to exploit sharing opportunities by caching distributed relations – which are translated into physical plans for execution.

First, we see that the three queries can share the scan of the **employees**, **departments** and **salaries** relations. Hence, a simple approach to work sharing would be to inject a cache operator in Query 1, which would steer the system to serve input relations from RAM instead of reading them from disk, when executing Query 2 and 3. A more refined approach could be to find common work (not only common I/O), in the form of *similar subexpressions* (SE) among the queries from the example, such as filtering and projecting records, and materialize intermediate results in RAM, so that to re-use such intermediate relations.

Figure 1 illustrates four examples of similar SEs, which are labelled as $\psi_i, i = 1, 2, 3, 4$ (we explain the meaning of this label in the next section). For example, consider the subexpression labelled as ψ_2: all three queries share the same sub-tree structure, in the form $Project_p(Filter_f(\textbf{employees}))$, but use different filtering predicates and projections. In principle, it is thus possible to save reading, parsing, filtering and projecting costs on the **employees** relation: by caching the intermediate output of a general form of subexpression, which subsumes the three similar sub-trees in each query. Such costs would be payed once, and the cached intermediate relation could serve three *consumer* queries. To this aim, we need to build a *covering expression* (CE) that combines the variants of the predicates appearing in the operators, e.g., considering ψ_2 the corresponding CE could be:

$$Project_{\text{id, name, dep, age}}(Filter_{\text{gender=F} \vee \text{age>30}}(\textbf{employees}))$$

Similarly, the SEs labelled as ψ_3 and ψ_4 share the projection and filtering on **department** and **salaries** relations.

We anticipate that, in the context of our work, it is possible to rank some SEs according to the benefits they bring, in terms of reducing redundant work.

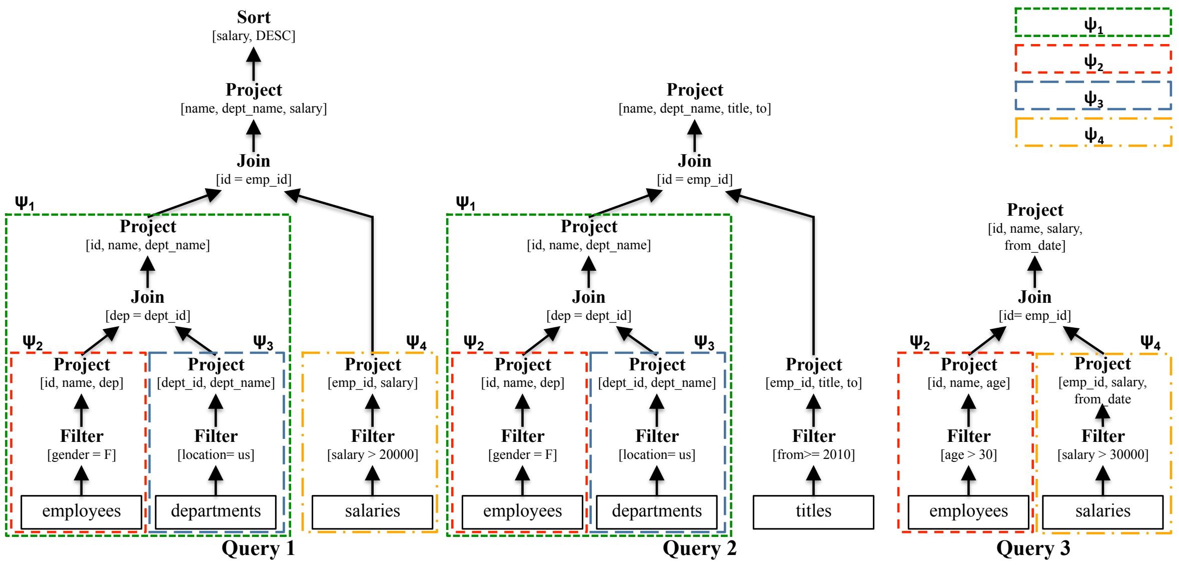

Figure 1: Logical plans for the queries in our running example. Similar subexpressions (SE) are emphasized by dashed boxes surrounding the corresponding sub-tree of each query. Boxes with the same border color denote the same SE.

For instance, the SE $Project_p(Filter_f(\textbf{employees}))$ leads to additional savings when compared to the SE $Filter_f(\textbf{employees})$, and caching the intermediate relation of the corresponding CE results in a smaller memory footprint because of its selectivity. More to the point, we now consider the SE labelled as ψ_1 in Figure 1: Query 1 and 2 share a common sub-tree in their respective logical plans, that involves selections, projections and joins. In this case, selecting this SE as a candidate to build a CE between Query 1 and 2 contributes to decreased scanning, computation and *communication costs*. However, since caching a relation in RAM bears its own costs and must satisfy capacity constraints, materializing in RAM the output of the CE might reveal not beneficial after all: a join operator could potentially produce an intermediate relation too big to fit in RAM.

Overall, given an input query set, our problem amounts to explore a potentially very large search space, to identify SEs, to build the corresponding CEs – which we also call *sharing plans*, and to decide which CEs to include in the optimized output plan. Our MQO strategy aims at reducing the search space to build CEs by appropriately pruning SEs according to their rank. Furthermore, a cost-based selection of candidate CEs must ensure memory budget constraints to be met.

4 CACHE-BASED WORK SHARING

We consider a set of concurrent queries submitted by multiple users to be parsed, analyzed and individually optimized by a query optimizer. Our MQO method operates on a set of optimized logical plans corresponding to the set of input queries, that we call the *input set*. We approach the MQO problem with the following steps:

§4.1: Similar subexpressions identification. The goal is to identify all common subexpressions in the input set. Two (or more) operators sharing similar subexpressions constitute a SE, which are candidates for building covering expressions (CEs).

§4.2: Building sharing plans. The goal is to construct one or more groups of covering expressions CEs for a set of SEs.

§4.3: Sharing plan selection. The goal is to select the best combination of CEs, using estimated costs and memory constraints. We model this step as a Multiple-Choice Knapsack problem, and use dynamic programming to solve it.

§4.4: Query rewriting. The last step to achieve MQO is to rewrite the input query set such as to use selected sharing plans.

4.1 Similar Subexpression Identification

Finding similar subexpressions, given an input set of logical plans, has received considerable attention in the literature. What sets our approach apart from previous works lies behind the very nature of the resource we use to achieve work sharing: memory is limited, and the overall MQO process we present is seen as a constrained optimization problem, which strives to use caching with parsimony. Thus, we use a general rule of thumb that prefers a large number of CEs (built from the corresponding SEs) with small memory footprints instead of a small number of CEs with large memory requirements. This is in line with low-level systems considerations: data materialization in RAM is *not* cost-free, and current parallel processing frameworks are sometimes fragile, when it comes to memory management under pressure.

Armed with the above considerations, we first consider the input to our task: we search SEs given a set of "locally optimized" query plans, which are represented in a tree form. Such input plans have been optimized by applying common rules such as early filtering, predicate push-down, plan simplification and collapsing [19]. The natural hierarchy of optimized logical plans implies that the higher in the tree an operator is, the less the data flowing from its edges. Hence, similar subexpressions that are found higher in the plan are preferred because they potentially exhibit smaller memory footprints.

Additional considerations are in order. Some operators produce output relations that are not easy to materialize in RAM: for example, binary operators such as join, generally produce large outputs that would deplete memory resources if cached. When searching for SEs, we recognize "cache unfriendly" operators and preempt them for being considered as valid candidates, either by selecting SEs that appear lower in the logical plan hierarchy (e.g., which could imply caching the input relations of a join), or by selecting SEs that subsume them (e.g., which could imply caching a relation resulting from filtering a join output). Currently, we treat the join, Cartesian product and Union as "cache unfriendly" operators. This means that our method does not produce SEs *rooted at* cache unfriendly operators; moreover, cache unfriendly operators can be shared inside a common SE only when they are syntactically equal.[1] Next, we provide the necessary definitions that are then used to describe the identification of SEs.

Definition 4.1 (Sub-tree). Given a logical plan of a query Q, represented as a tree τ^Q where leaf nodes are base relations and each intermediate node is a relational algebra operator, a sub-tree τ_s^Q of τ^Q is a continuous portion of the logical plan of Q containing an intermediate node of τ^Q and all its descendant in τ^Q. In other words, a sub-tree includes all the base relations and operators that are necessary to build its root.

If the context is clear, we denote a sub-trees simply as τ, without indicating from which query it has been derived. Given any two sub-trees, we need to determine if they have the same *structure* in terms of base relations and operators. To this aim, we define a similarity function based on a modified Merkle Tree (also known as *hash tree*)[44], whereby each internal node identifier is the combination of identifiers of its children. In particular, given an operator u, its identifier, denoted by ID(u), is given by:

$$\mathrm{ID}(u) = \begin{cases} (u.label) & u \in \{\text{filter, project,input rel.}\} \\ (u.label, u.attributes) & \text{otherwise.} \end{cases}$$

Notice that this definition makes a distinction between loose and strict identifier. A loose identifier, such that used for projections and selections, allows the construction of a *shared operator* that subsumes the individual attributes with more general ones, which allows sharing computation among SEs. Instead, a strict identifier, such that used for all other operators (including joins and unions), imposes strict equality for two sub-graphs to be considered SEs. In principle, this restricts the applicability of a shared operator. However, given the above considerations about cache unfriendly operators, our approach still shares both I/O and computation.

Definition 4.2 (Fingerprint). Given a sub-tree τ, its *fingerprint* is computed as

$$\mathcal{F}(\tau) = \begin{cases} h(\mathrm{ID}(\tau_{\text{root}})) & \tau_{\text{root}} = \text{leaf} \\ h(\mathrm{ID}(\tau_{\text{root}})|\mathcal{F}(\tau_{\text{child}})) & \tau_{\text{root}} = \text{unary} \\ h(\mathrm{ID}(\tau_{\text{root}})|\mathcal{F}(\tau_{\text{l.child}})|\mathcal{F}(\tau_{\text{r.child}})) & \tau_{\text{root}} = \text{binary} \end{cases}$$

where $h()$ is a robust cryptographic hash function, and the operation | indicates concatenation.

The fingerprint $\mathcal{F}(\tau)$ is computed recursively starting from the root of the sub-tree (τ_{root}), down to the leaves (that is, input relations). If the root is a unary operator, we compute the fingerprint of its child sub-tree (τ_{child}), conversely in case of a binary operator, we consider the left and right sub-trees ($\tau_{\text{l.child}}$ and $\tau_{\text{r.child}}$). For the sake of clarity, we omit an additional sorting which ensures the isomorphic property for binary operators: for example, *TableA join TableB* and *TableB join TableA* are two isomorphic expressions, and have the same fingerprint.

We are now ready to define what a *similar subexpression* is.

[1]Our method can be easily extended for sharing similar join operators, for example by applying the "equivalence classes" approach used in [5]. Despite technical simplicity, our current optimization problem formulation would end-up discarding such potential SEs, due to their large memory footprints. Hence, we currently preempt such SEs from being considered.

Algorithm 1 Similar subexpressions identification

Input: Array of logical plans (trees), threshold k
Output: Set S of SEs ω_i

```
1:  procedure IdentifySEs([$\tau^{Q_1}$, $\tau^{Q_2}$, ...$\tau^{Q_N}$])
2:      FT ← $\emptyset$
3:      foreach $\tau \in [\tau^{Q_1}, \tau^{Q_2}, \dots \tau^{Q_N}]$ do
4:          nodeToVisit ← Add($\tau$)
5:          while nodeToVisit not empty do
6:              $\tau^{\text{curr}}$ ← Pop(nodeToVisit)
7:              $\psi$ ← $\mathcal{F}(\tau^{\text{curr}})$
8:              if CacheFriendly($\tau^{\text{curr}}_{\text{root}}$) then
9:                  FT.AddValueSet($\psi$, $\tau^{\text{curr}}$)
10:             end if
11:             if (! CacheFriendly($\tau^{\text{curr}}_{\text{root}}$) $\vee$
12: ContainsUnfriendly($\tau^{\text{curr}}$)) then
13:                 nodeToVisit ← Add($\tau^{\text{curr}}_{\text{children}}$)
14:             end if
15:         end while
16:     end for
17:     $S$ ← $\emptyset$
18:     foreach $\psi \in$ FT.Keys do
19:         if |FT.GetValue($\psi$)| $\geq k$ then
20:             $S$ ← $S \cup$ FT.GetValue($\psi$)
21:         end if
22:     end for
23:     return $S$
24: end procedure
```

Definition 4.3 (Similar subexpression). A similar subexpression (SE) ω is a set of sub-trees that have the same fingerprint ψ, *i.e.* $\omega = \{\tau_i \mid \mathcal{F}(\tau_i) = \psi\}$.

Algorithm 1 provides a pseudo-code of our procedure to find, given a set of input queries, the SEs according to Definition 4.3 that will be the input of the next phase, the search for covering expressions. The underlying idea is to avoid a brute-force search of fingerprints, which would produce a large number of SEs. Instead, by proceeding in a top-down manner when exploring logical plans, we produce fewer SEs candidates, by interrupting the lookup procedure as early and as "high" as possible.

The procedure uses a fingerprint table FT (line 2) to track SEs: this is a HashMap, where the key is a fingerprint ψ, and the value is a set of subtrees. Each logical plan from the input set of queries is examined in a depth-first manner. We first consider the whole query tree (line 4) and check if its root is a cache-friendly operator: in this case, we add the tree to the SEs identified by its fingerprint. The method AddValueSet(ψ, τ) retrieves the value (which is a set) from the HashMap FT given the key ψ (line 9), and adds the subtree τ to such a set – if the key does not exists, it adds it and create a value with a set containing the subtree τ. If the root is not a cache-friendly operator, or the logical plan contains a cache-unfriendly operator, then we need to explore the subtrees (line 13), *i.e.* we consider the root's child (if the the operator at the root is unary) or children (otherwise).

At the end, we extract the set of SEs from the HashMap FT: we consider the SEs bigger than a threshold k in order to focus on SEs that offer potential work sharing opportunities.

Going back to our running example, Algorithm 1 outputs a set of SEs as follows $\{\omega_1, \omega_2, \omega_3, \omega_4\}$ – in Figure 1 the sub-trees corresponding to them are labelled ψ_1, ψ_2, ψ_3 and ψ_4, where ψ_i is the fingerprint of SE ω_i. For instance, ω_1 contains two sub-trees (one from Query 1, and one from Query 2), while ω_2 contains three sub-trees, one from each query.

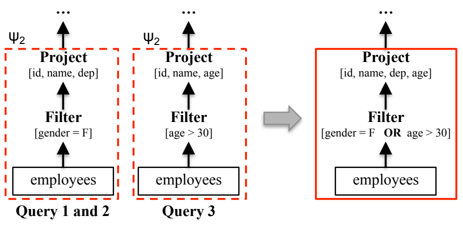

Figure 2: Building covering expression example. The first and second trees are two similar subexpressions. The third tree is the covering subexpression.

4.2 Building Sharing Plans

Given a list of candidate SEs, this phase aims at building covering subexpressions (CEs) corresponding to identified SEs, and generate a set of candidate groups of CEs for their final selection.

Covering subexpressions. For each similar sub-query in the same SE ω_i, the goal is to produce a new plan to "cover" all operations of each individual sub-query. Recall that all sub-trees τ_j within a SE ω_i share the same sub-query plan fingerprint: that is, they operate on the same input relation(s) and apply the same relational operators to generate intermediate output relations. If the operator attributes are exactly the same across all τ_j, then the CE will be identical to any of the τ_j. In general, however, operators can have different attributes or predicates. In this case, the CE construction is slightly more involved.

First, we note that, by construction, the only shared operators we consider are projections and selections. Indeed, for *cache unfriendly* operators, the SE identification phase omits their fingerprint from the lookup procedure (see Algorithm 1, lines 8-9). Nevertheless, they could be included within a subtree, but they are in any case "surrounded" by cache-friendly operators (see for instance in Figure 1, the SE labeled as ψ_1). As a consequence, a CE can be constructed in a top-down manner, by "OR-ing" the filtering predicates and by "unioning" the projection columns of the corresponding operators in the SE. The CE thus produces and materializes all output records that are needed for its consumer queries. Figure 2 illustrates an example of CE for a simple SE of two sub-queries taken from the running example shown in Figure 1. In particular, we consider the SE labeled as ψ_2.

The resulting CE contains the same operators as the subtrees $\tau_j \in \omega_i$, but with modified predicates or attribute lists.

In general, we can build a CE, which we denote with Ω_i, from a SE ω_i, by applying a transformation function $f()$, $[\tau_1, ...\tau_m] \xrightarrow{f()} \tau_i^*$, which trasforms a collection of similar sub-trees to a single, *covering* sub-tree τ_i^*. Note that the resulting covering sub-tree has the fingerprint of the sub-trees in ω_i.

Definition 4.4 (Covering subexpression). A Covering subexpression (CE) $\Omega_i = f(\omega_i)$ is a sub-tree τ_i^* derived from the SE ω_i by applying the transformation $f()$, with $\mathcal{F}(\tau_i^*) = \mathcal{F}(\tau_j) \forall \tau_j \in \omega_i$, such that all $\tau_j \in \omega_i$ can be derived from τ_i^*.

In summary, the query plan τ_i^* that composes Ω_i contains the same nodes as any subtree $\tau_j \in \omega_i$, changing the predicates of the selections (OR of all the predicates in τ_j) and projections (union of all the predicates in τ_j).

Once the set of CEs, $\Omega = \{\Omega_1, \Omega_2, \dots\}$, has been derived from the corresponding set of SEs, $\omega = \{\omega_1, \omega_2, \dots\}$, we need to face the problem of CE *selection*. The main question we need to answer is: among the CEs contained in the set Ω, which ones should be cached? Each CE covers different portions of the query logical plans, therefore a CE may include another CE. Looking at the running example shown in Figure 1, we have that Ω_1 (derived from ω_1, in the figure labeled with ψ_1) contains Ω_3 (derived from ω_3 and labeled in the figure with ψ_3). If we decide to store Ω_1 in the cache, it becomes questionable to store Ω_3 as well.

The next step of our process is then to identify the potential combinations of *mutually exclusive* CEs that will be the input of the optimization problem: each combination will have a *value* and *weight*, where the value provides a measure of the work sharing opportunities, and the weight indicates the amount of space required to cache the CE in RAM. We start considering how to compute such values and weights, and we proceed with the algorithm to identify the potential combination of CEs.

CE value and weight: a cost-based model. We use cardinality estimation and cost modeling to reason about the benefit of using CEs. The objective is to estimate if a given CE, that could serve multiple consumer queries, yields lower costs than executing *individually* the original queries it subsumes.

The *cardinality estimator* component analyzes relational operators to estimate their output size. To do so, it first produces statistics about input relations. At relation level, it obtains the number of records and average record size. At column level, it collects the min and max values, approximates column cardinality and produces an equi-width histogram for each column.

The *cost estimator* component uses the results from cardinality estimation to approximate a (sub) query execution cost. We model the total execution cost of a (sub) query as a combination of CPU, disk and network I/O costs. Hence, given a sub-tree τ_j, we denote by $C_E(\tau_j)$ the execution cost of sub-tree τ_j. This component recursively analyzes, starting from the root of sub-tree τ_j, relational operators to determine their cost (and their selectivity), which is the multiplication between predefined constants (representative of the compute cluster running the parallel processing framework) and the estimated number of input and output records. Given a SE $\omega = \{\tau_1, \tau_2, \dots, \tau_m\}$, the total execution cost $C(\omega_i)$ related to the execution of all similar sub-trees $\tau_j \in \omega_i$ without the work-sharing optimization is given by

$$C(\omega_i) = \sum_{j=1}^{m} C_E(\tau_j). \tag{1}$$

Instead, the cost of using the corresponding CE Ω_i must account for both the execution cost of the common sub-tree τ_i^*, and materialization (C_W) and retrieving (C_R) costs *associated to the cache operator we use in our approach*, which accounts for write and read operations:

$$C(\Omega_i) = C_E(\tau_i^*) + C_W(|\tau_i^*|) + m \cdot C_R(|\tau_i^*|), \tag{2}$$

where both $C_W(|\tau_i^*|)$ and $C_R(|\tau_i^*|)$ are functions of the cardinality $|\tau_i^*|$ of the intermediate output relation obtained by executing τ_i^*. Eq. 2 indicates that retrieving costs are "payed" by each of the m *consumer* queries from the SE ω_i that can use the CE Ω_i.

Then, we can derive the *value* of a CE Ω_i, denoted by $v(\Omega_i)$, as the difference between the cost of an unoptimized set of sub-trees (execution of ω_i) and the cost of the CE Ω_i:

$$v(\Omega_i) = C(\omega_i) - C(\Omega_i). \tag{3}$$

From Equations 1 and 2, we note that $v(\Omega_i)$ is an increasing function in m. Indeed, the more similar sub-queries a CE can serve, the higher its value.

Along with the value, we need to associate to a CE also a *weight*, since the memory is limited and we need to take into

Algorithm 2 Algorithm to generate CE candidates.

```
Input: Set $\Omega$ of CEs
Output: Set of Knapsack items (potential CEs)
 1: procedure GenerateKPitems($\Omega = \{\Omega_1, \Omega_2, \dots\}$)
 2:     $\Omega^{\text{exp}} \leftarrow \emptyset$
 3:     while $\Omega$ not empty do
 4:         $\Omega_i \leftarrow$ PopLargest($\Omega$)
 5:         DescSet $\leftarrow$ FindDescendant($\Omega_i$, $\Omega$)
 6:         Group$_i \leftarrow [\Omega_i] \cup$ Expand(DescSet)
 7:         $\Omega^{\text{exp}} \leftarrow \Omega^{\text{exp}} \cup \{\text{Group}_i\}$
 8:         Remove(DescSet, $\Omega$)
 9:     end while
10:     return $\Omega^{\text{exp}}$
11: end procedure
```

account if a CE can fit in the cache. The weight, denoted by $w(\Omega_i)$ is the size required to cache in RAM the output of Ω_i, *i.e.* $w(\Omega_i) = |\tau_i^*| \stackrel{\Delta}{=} |\Omega_i|$.

Having defined the CE value and weight, we describe next the algorithm to identify the potential combination of CE.

Generating the candidate set of CEs. Next, we focus on the problem of generating a combinatorial set of CEs, with their associated value and weight, to be given as an input to the multi-query optimization solver we have designed. Given the complexity of the optimization task, our goal is to produce a small set of valuable alternative options, which we call the candidate set of CEs. We present an algorithm to produce such a set, but first illustrate the challenges it addresses using the example shown in Fig. 1.

Let's focus on CE Ω_1 (corresponding to the sub-trees labeled as ψ_1). A naive enumeration of all possible choices of candidate CE to be cached leads to the following, *mutually exclusive* options: (i) Ω_1, (ii) Ω_2, (iii) Ω_3, (iv) both (Ω_2,Ω_3), (v) both (Ω_1,Ω_2), and (vi) both (Ω_1,Ω_3). Intuitively, however, it is easy to discern valuable from wasteful options. For example, the compound CE (Ω_1, Ω_2) could be a good choice, since Ω_2 can be cached to serve query 1 and 2 – and of course used to build Ω_1 – and for query 3. Conversely, caching the compound (Ω_1, Ω_3) brings less value, since it only benefits query 1 and query 2, but costs more than simply caching Ω_1, which also serves both query 1 and 2.

It is thus important to define *how to compute the value and weight of compound* CE. In this work we only consider compound CEs for which value and weight are *additive* in the values and weights of their components. This is achieved with compounds of *disjoint* CEs, *i.e.*, those that have no common sub-trees.

For example, consider the two CEs Ω_1 and Ω_2, and the sub-trees used to build them. The CE Ω_2 is included in Ω_1, but only some of the originating sub-trees of Ω_2 are included in the originating sub-trees of Ω_1 (in particular, the ones in query 1 and 2, but not in query 3). Given our definition of the value and the weight of CEs, the value and the weight of the compound (Ω_1, Ω_2) may not be equal to the sums of the values and of the weights of each individual CE, since part of the CE need to be reused to compute different sub-trees. Thus, we discard this option from the candidate set.

Algorithm 2 generates the candidate input for the optimization solver as a set of non-overlapping groups of CEs; then, the optimization algorithm selects a single candidate for each group in order to determine the best set of CEs to store in memory. Given the full set of Ω of CEs as input, we consider CE Ω_i starting from the root of the logical plan and remove it from the set (line 4). We then look for its descendants from the input set Ω, *i.e.* all the CEs contained in Ω_i (line 5). With a CE and its descendant, we build a list of options that contains (i) the CE itself and its individual descendants, and (ii) all the compounds of *disjoint* descendant CEs (line 6 and 7). We then remove the descendant from Ω and continue the search for other groups.

Considering our running example, we start from $\Omega = \{\Omega_1, \Omega_2, \Omega_3, \Omega_4\}$. The "largest" CE is Ω_1, and its descendants are Ω_2 and Ω_3, therefore the list of mutually exclusive options for this group would be $[\Omega_1, \Omega_2, \Omega_3, (\Omega_2, \Omega_3)]$. The output of Alg. 2 then is:

$$\{[\Omega_1, \Omega_2, \Omega_3, (\Omega_2, \Omega_3)], [\Omega_4]\}, \tag{4}$$

where the notation $(\cdot, \cdot)$ indicates a compound CE, and $[\cdot, \cdot]$ indicates a group of related CEs.

Note that a CE may be part of more than one larger CE: to keep the algorithm simple, we consider only the largest ancestor for each CE. To each option, we associate the value and the weight (in case of a compound, the sum of each component), that will be used by the optimization solver.

4.3 Sharing Plan Selection

Next, we delve into our MQO problem formulation. In this work, we model the process that selects which sharing plan to use as a Multiple-choice Knapsack problem (MCKP) [45]. Essentially, the knapsack contains items (that is, sharing plans or CEs) that have a *weight* and a *value*. The knapsack capacity is constrained by a constant c: this is representative of the memory constraints given to the work sharing optimizer. Hence, the sum of the weights of all items placed in the knapsack cannot exceed its capacity c.

Our problem is thus to select which set of CEs (single, or compound) to include in the knapsack. The output of the previous phase (and in particular, the output of Algorithm 2) is a set containing m groups of mutually exclusive options, or items. Each group G_i, $i = 1, 2, \dots, g$, contains $|G_i|$ items, which can be single CE or compounds of CEs. For instance, looking at our running example, the output shown in Eq. (4) contains $g = 2$ groups: the first group has 4 items, the second group just one item. Given a group i, each item j has a value $v_{i,j}$ and a weight $w_{i,j}$ computed as described in Sect. 4.2.

The MCKP solver needs to choose *at most* one item from each group such that the total value is maximized, while the corresponding total weight must not exceed the capacity c. More formally, the problem can be cast as following:

$$\begin{aligned} &\text{Maximize} \sum_{i=1}^{g} \sum_{j=1}^{|G_i|} v_{i,j} x_{i,j} \\ &\text{subject to} \sum_{i=1}^{g} \sum_{j=1}^{|G_i|} w_{i,j} x_{i,j} \le c \\ &\sum_{j=1}^{|G_i|} x_{i,j} \le 1, \forall i = 1 \dots g \\ &x_{i,j} \in \{0, 1\}, \forall i = 1 \dots g, j = 1 \dots |G_i| \end{aligned} \tag{5}$$

where the variable $x_{i,j}$ indicates if item j from group i has been selected or not. The MCKP is a well-known NP-Hard problem: in this work, we implement a dynamic programming technique to solve it [46].

Note that alternative formulations exist, for which a provably optimal greedy algorithm can be constructed: for example, we could consider a fractional formulation of the knapsack problem. This approach, however, would be feasible only if the underlying query execution engine could support partial caching of a relation.

As it turns out, the system we target in our work does support hierarchical storage levels for cached relations: what does not fit in RAM, is automatically stored on disk. Although this represents an interesting direction for future work (as it implies a linear time greedy heuristic can be used), in this paper we limit our attention to the 0/1 problem formulation.

4.4 Query Rewriting

The last step is to transform the original input queries to benefit from the selected combination of *cache plans*.

Recall that the output of a *cache plan* is materialized in RAM after its execution. Then, for each input query that is a *consumer for a given cache plan*, we build an *extraction plan* which manipulates the cached data to produce the output relation, as it would be obtained by the original input query. In other words, in the general case, we apply the original input query to the cached relation instead of using the original input relation. In the case of a CE subsuming identical SEs, the extraction plan is an identity: the original query simply replaces the sub-tree containing the CE by its cached intermediate relation. Instead, if shared operators are used – because of SEs having the same fingerprint but different attributes – we build an extraction plan that applies the original filter and projection predicates or attributes to “extract” relevant tuples from the cached relation produced from the CE.

Considering our running example, assume that the output of the MCKP solver is to store Ω_2 and Ω_3 in cache. Ω_3 derives from ω_3, where the composing sub-trees (one from query 1, and one from query 2) are the same, therefore the extraction plan will be Ω_3 itself. Instead, ω_2 (from which Ω_2 derives) contains sub-trees with different filtering and projection predicates: when Ω_2 is materialized in the cache, we need to apply the correct filtering (e.g., “gender = F”) and projection predicates to extract the actual result when considering the different queries.

5 EXPERIMENTAL EVALUATION

We now present experimental results to evaluate the effectiveness of our methodology, which we implement for the Apache Spark and SparkSQL systems – the details of the implementation can be found in our companion TR [48]

Experimental setup. We run our experiments on a cluster consisting of 8 server-grade worker nodes, with 8 cores each and a 1 Gbps commodity interconnect. Each worker is granted 30 GB of RAM each, of which half is dedicated to caching. We use the queries in the TPC-DS benchmark library for Spark SQL developed by Databricks [47], and generate a CSV dataset with scaling factor of 50. We use Apache Spark 2.0: for all test, we clear the operating system’s buffer cache in all workers and master, and disable the “data compression" feature of Spark.

Results. We select a subset of all queries available in the TPC-DS benchmark, and focus on the 50 queries that can be successfully executed without failures or parsing errors. We present results for a setup in which we consider all the 50 queries and execute them in the order of their identifiers. Figure 3 shows the empirical Cumulative Distribution Function (CDF) of the runtime ratios between a system absorbing the workload with MQO enabled and disabled. Overall, we note that, for 60% of the queries, we obtain a 80% decrease of the runtime. In total, our approach reduces the runtime for 82% of the queries. On the other hand, 18% of the queries experience a larger runtime, which is explained by the overheads associated to caching. Overall, our optimizer has identified 60 SEs, and it has built 45 CEs. The cache used to store the output is approximately 26 GB (out of 120 GB available). The optimization process took less than 2 seconds, while the query runtime are in the order of tens of minutes (individually) and hours (all together).

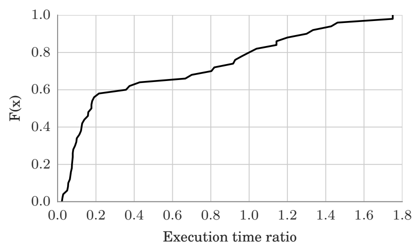

Figure 3: CDF of the performance gains of worksharing for a TCP-DS workload consisting of 50 selected queries.

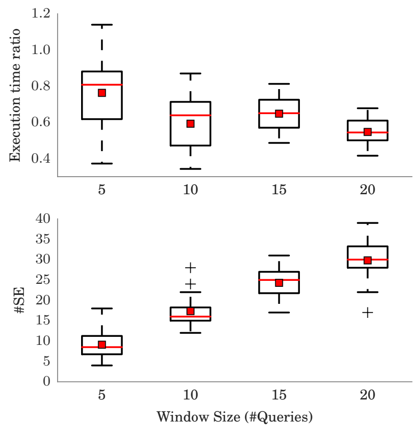

Figure 4: Execution time ratio and number of similar subexpression within a group of queries (given by the size of the window) as the window size increases.

Next, we consider an experimental setup in which we emulate the presence of a queuing component that triggers the execution of our worksharing optimization. In particular, since TPC-DS queries have no associated submission timestamp, we take a randomized approach (without replacement) to select which queries are submitted to the queuing component, and parametrize the latter with the number of queries – we call this parameter the *window size* – to accumulate before triggering our MQO mechanism. For a given window size, we repeat the experiment, *i.e.*, we randomly select queries from the full TPC-DS workload, 20 times, and we build the corresponding empirical CDF of the runtime ratio, as defined above. We also measure the number of SEs identified within the window size, and show the corresponding empirical CDF. Given this setup, we consider all possible combinations of queries to assess the benefits of worksharing.

Figure 4 shows the boxplots of the runtime ratio (top) and number of similar subexpression identified (bottom) for different window sizes. The boxplots indicate the main percentiles (5%, 25%, 50%, 75%, 95%) of the empirical CDF, along with the average (red lines). The Figure shows a clear pattern: as the size of the window increases, there are more chances of finding a high number of SE, thus better sharing opportunities, which translates into reduced aggregate runtime. We observe a 20% decrease of the aggregate runtime (median) with a window size of only five queries, which ramps up to 45% when the window size is set to 20 queries.

Note that a queuing mechanism can introduce an additional delay for the execution of a query, because the system needs to accumulate a sufficient number of queries in the window before triggering their optimization and execution. Investigating the trade-off between efficiency and delay, as well as studying scheduling policies to steer system behavior is part of our future research agenda. Due to space contraints, we refer to [48] for additional evaluations and discussion.

6 CONCLUSION

We presented a new approach to MQO that uses in-memory caching to improve the efficiency of computing frameworks such as Apache Spark. Our method takes a batch of input queries and finds common (sub)expressions, leading to the construction of covering expressions that subsume the individual work required by each query. To make the search problem tractable, we used several techniques: modified hash trees to quickly identify common sub-graphs, and an algorithm to enumerate (and prune) feasible common expressions. MQO was cast as a multiple-choice knapsack problem: each feasible common expression was associated with a value (representative of how much work could be shared among queries) and a weight (representative of the memory pressure imposed by caching the common data), and the goal was to fill a knapsack representative of memory constraints.

To quantify the benefit of our approach, we implemented a prototype for Apache Spark SQL, and we used well-known workloads. Our results indicated that worksharing opportunities are frequent, and that our method brings substantial benefits in terms of reduced query runtime, with up to an 80% reduction for a large fraction of the submitted queries.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Comm. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *ACM SIGOPS Op. Systems Review*, vol. 41, no. 3, 2007, pp. 59–72.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of USENIX NSDI*, 2012, pp. 2–2.

[4] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters." in *Proc. of OSDI*, vol. 10, 2010, pp. 1–8.

[5] J. Zhou, P.-A. Larson, J.-C. Freytag, and W. Lehner, "Efficient exploitation of similar subexpressions for query processing," in *Proc. of ACM SIGMOD*, 2007, pp. 533–544.

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

[7] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *ACM SIGMOD Record*, vol. 29, no. 2, 2000, pp. 249–260.

[8] J. Goldstein and P.-Å. Larson, "Optimizing queries using materialized views: a practical, scalable solution," in *ACM SIGMOD Record*, vol. 30, no. 2, 2001, pp. 331–342.

[9] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham, "Materialized view selection and maintenance using multi-query optimization," in *ACM SIGMOD Record*, vol. 30, no. 2, 2001, pp. 307–318.

[10] N. N. Dalvi, S. K. Sanghai, R. Parsan, and S. Sudarshan, "Pipelining in multi-query optimization," in *Proc. of ACM PODS*, 2001, pp. 59–70.

[11] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki, "Sharing data and work across concurrent analytical queries," *VLDB Endowment*, vol. 6, no. 9, pp. 637–648, Jul. 2013.

[12] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "Qpipe: A simultaneously pipelined relational query engine," in *Proc. of ACM SIGMOD*, 2005, pp. 383–394.

[13] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez, "The datapath system: A data-centric analytic processing engine for large data warehouses," in *Proc. of ACM SIGMOD*, 2010, pp. 519–530.

[14] G. Giannikis, G. Alonso, and D. Kossmann, "Shareddb: Killing one thousand queries with one stone," *VLDB Endowment*, vol. 5, no. 6, pp. 526–537, Feb. 2012.

[15] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare2: Sharing across multiple queries in mapreduce," *VLDB Endowment*, vol. 3, no. 1-2, pp. 494–505, Sep. 2010.

[16] G. Wang and C.-Y. Chan, "Multi-query optimization in mapreduce framework," *VLDB Endowment*, vol. 7, no. 3, pp. 145–156, Nov. 2013.

[17] Y. N. Silva, P.-A. Larson, and J. Zhou, "Exploiting common subexpressions for cloud query processing," in *Proc. of IEEE ICDE*, 2012, pp. 1337–1348.

[18] A. El-Helw, V. Raghavan, M. A. Soliman, G. Caragea, Z. Gu, and M. Petropoulos, "Optimization of common table expressions in mpp database systems," *VLDB End.*, vol. 8, no. 12, pp. 1704–1715, 2015.

[19] M. Armbrust, et al., "Spark sql: Relational data processing in spark," in *Proc. of ACM SIGMOD*, 2015, pp. 1383–1394.

[20] S. Finkelstein, "Common expression analysis in database applications," in *Proc. of ACM SIGMOD*, 1982, pp. 235–245.

[21] J. Shim, P. Scheuermann, and R. Vingralek, "Dynamic caching of query results for decision support systems," in *Proc. of IEEE SSDBM*, 1999, pp. 254–.

[22] G. Candea, N. Polyzotis, and R. Vingralek, "Predictable performance and high query concurrency for data analytics," *The VLDB Journal*, vol. 20, no. 2, pp. 227–248, Apr. 2011.

[23] ——, "A scalable, predictable join operator for highly concurrent data warehouses," *VLDB Endowment*, vol. 2, no. 1, pp. 277–288, Aug. 2009.

[24] J. Yang, K. Karlapalem, and Q. Li, "Algorithms for materialized view design in data warehousing environment," in *VLDB*, vol. 97, 1997, pp. 25–29.

[25] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in sql databases." in *VLDB*, vol. 2000, 2000, pp. 496–505.

[26] X. Baril and Z. Bellahsene, "Selection of materialized views: A cost-based approach," in *Advanced Information Systems Engineering*. Springer, 2003, pp. 665–680.

[27] C. Zhang and J. Yang, "Genetic algorithm for materialized view selection in data warehouse environments," in *DataWarehousing and Knowledge Discovery*. Springer, 1999, pp. 116–125.

[28] R. Derakhshan, F. K. Dehne, O. Korn, and B. Stantic, "Simulated annealing for materialized view selection in data warehousing environment." in *Databases and applications*, 2006, pp. 89–94.

[29] P. Kalnis, N. Mamoulis, and D. Papadias, "View selection using randomized search," *Data & Knowledge Engineering*, vol. 42, no. 1, pp. 89–111, 2002.

[30] P. Agrawal, D. Kifer, and C. Olston, "Scheduling shared scans of large data files," *VLDB Endowment*, vol. 1, no. 1, pp. 958–969, 2008.

[31] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proc. of ACM SoCC*, 2011, p. 7.

[32] I. Elghandour and A. Aboulnaga, "Restore: reusing results of mapreduce jobs," *VLDB Endowment*, vol. 5, no. 6, pp. 586–597, 2012.

[33] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using mapreduce," in *Proc. of ACM SIGMOD*, 2011, pp. 985–996.

[34] ——, "Scalla: a platform for scalable one-pass analytics using mapreduce," *ACM Trans. on Database Systems*, vol. 37, no. 4, p. 27, 2012.

[35] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. Roy Chowdhury, "Reuse-based optimization for pig latin," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '16. ACM, 2016, pp. 2215–2220.

[36] L. L. Perez and C. M. Jermaine, "History-aware query optimization with materialized intermediate views," in *Proceedings of the 30th IEEE International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, 2014, pp. 520–531.

[37] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao, "Computation reuse in analytics job service at microsoft," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 191–203.

[38] A. Jindal, K. Karanasos, S. Rao, and H. Patel, "Selecting subexpressions to materialize at datacenter scale," *Proc. VLDB Endow.*, vol. 11, no. 7, pp. 800–812, Mar. 2018.

[39] T. Azim, M. Karpathiotakis, and A. Ailamaki, "Recache: Reactive caching for fast analytics over heterogeneous data," *VLDB Endowment*, vol. 11, no. 3, 2017.

[40] K. Dursun, C. Binnig, U. Cetintemel, and T. Kraska, "Revisiting reuse in main memory database systems," in *Proc. of ACM SIGMOD*, 2017, pp. 1275–1289.

[41] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J. Schmitz-Hermes, "Adaptive caching in big sql using the hdfs cache," in *Proc. of ACM SoCC*, 2016, pp. 321–333.

[42] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves, "An architecture for recycling intermediates in a column-store," *ACM Trans. on Database Systems*, vol. 35, no. 4, p. 24, 2010.

[43] F. Nagel, P. Boncz, and S. D. Viglas, "Recycling in pipelined query evaluation," in *Proc. of IEEE ICDE*, 2013, pp. 338–349.

[44] R. C. Merkle, "Protocols for public key cryptosystems," *IEEE Symposium on Security and Privacy*, p. 122, 1980.

[45] P. Sinha and A. A. Zoltners, "The multiple-choice knapsack problem," *Operations Research*, vol. 27, no. 3, pp. 503–515, 1979.

[46] H. Kellerer, U. Pferschy, and D. Pisinger, *Introduction to NP-Completeness of knapsack problems*. Springer, 2004.

[47] "Spark sql performance test," https://github.com/databricks/spark-sql-perf.

[48] P. Michiardi, D. Carra, and S. Migliorini, "Cache-based multi-query optimization for data-intensive scalable computing frameworks," *arXiv preprint arXiv:1805.08650*, 2018.